PART FOUR. MISCELLANEOUS WORK

XIV. A POLYPHONIC ELECTRONIC ORGAN

1. SPATIAL SEPARATION OF VOICES IN POLYPHONIC MUSIC

In polyphonic music of the Baroque period, many seemingly independent voices are interwoven together. These voices would carry the thematical phrases alternatively or in succession. This type of music played on keyboard instruments like organ or harpsichord is very difficult for the listener to identify and to follow the individual voices. This is particularly true in Bach's organ compositions. Many of Bach's organ work were transcribed to be played by an entire orchestra. Different voices can thus be separately played by different instruments and we can observed the effect of tonal separation of voices. The listener can thus follow a voice by following an instrument.

My experiment here is trying to separate the voices spatially by playing a polyphonic piece through many speakers placed around a large room. It is interesting to see how the listener can differentiate the voices by the spatial origins of the voices. This technique was demonstrated in many European churches which have several organs which can be played simultaneously from a single console. This electronic organ is designed to play up to 12 independently programmable voices through 12 speakers. It is thus very convenient to test the effects of spatially separated voices.

2. THE ELECTRONIC ORGAN

The electronic organ is controlled and programmed by an IBM compatible personal computer. A Parallel Interface Card is inserted into an I/O slot on PC-Bus to generate the voices. There are 4 Intel 8253A Counter-Timer chips on this card. Each 8253 has three counter-timer channels, which are programmed to operate in Mode 2 to generate square waves. The frequency of each counter-timer channel can be programmed by writing a 16 bit count word into the control register. The counters are driven by the 4.77 MHz system clock, divided by a pair of flip-flops to a base frequency of 1.19 MHz.

Fig. 27 shows the interface circuitry to the PC bus, including the divide by 4 clock, the I/O address decoder, and the data line buffer. Fig. 28 shows the circuit around one of the four 8253A's, and the power transistor with the speaker. The circuitry is very simple and can be wire-wraped on a proto-card. For all thumb hands, this Parallel Interface Card is available from

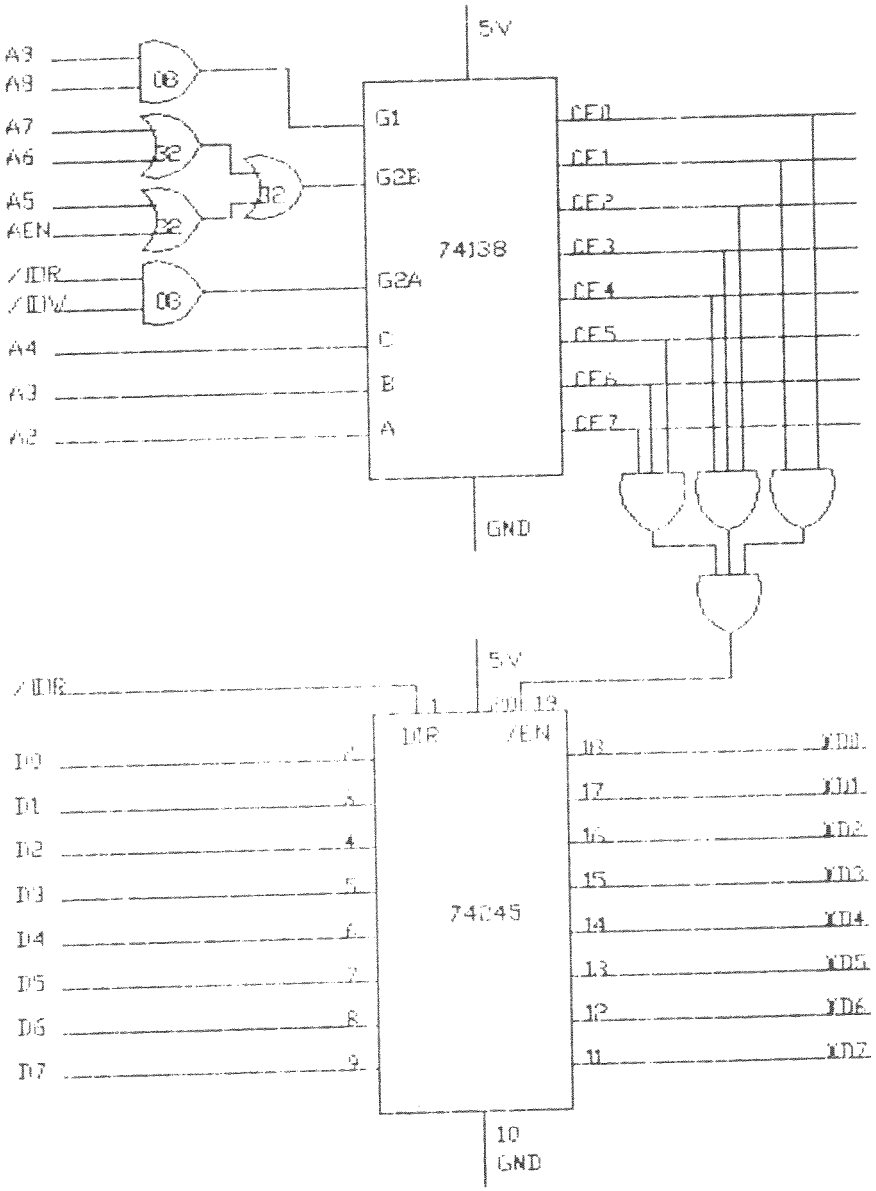


Figure 27. PC interface of the electronic organ

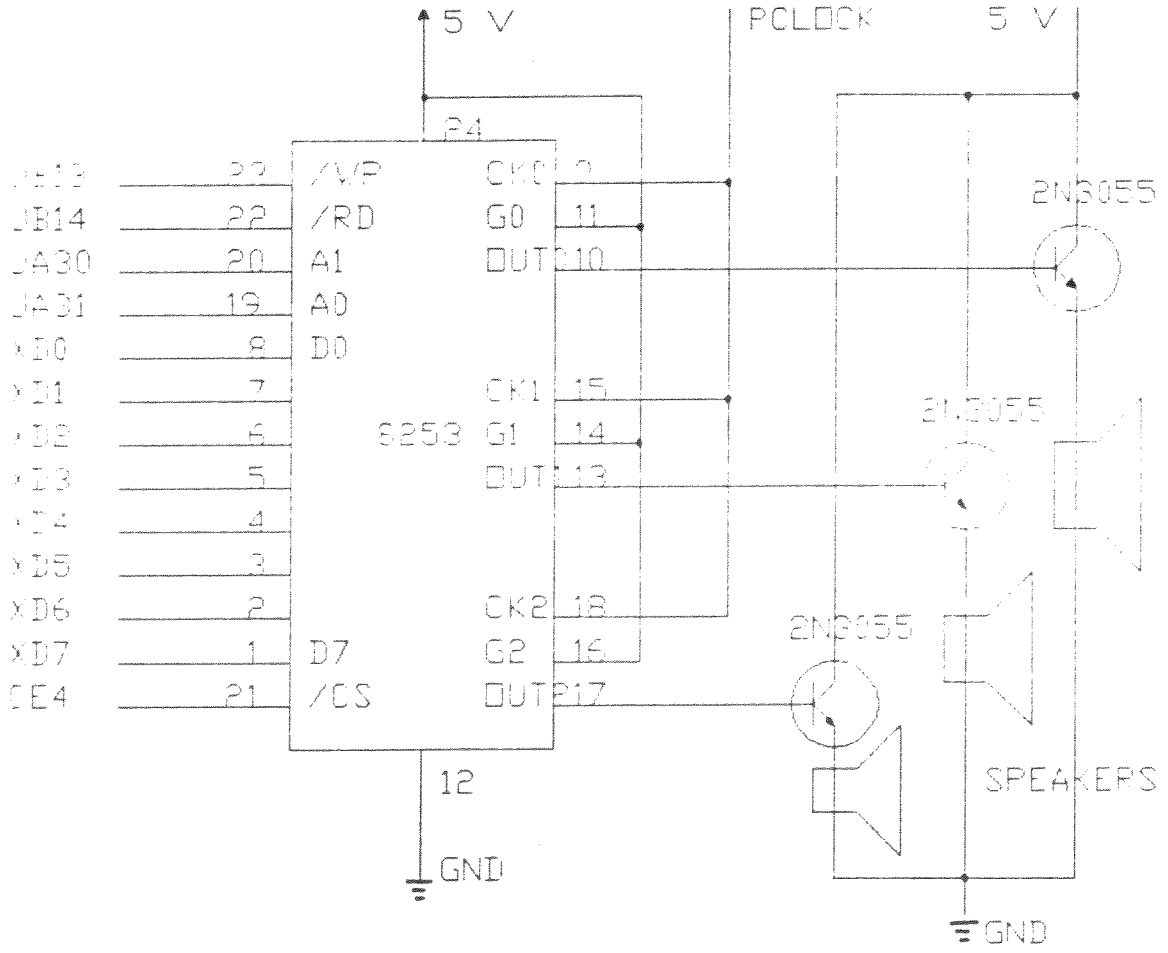


Figure 28. 8253 counter-timer for tone generation

The output of 8253A can only drive a small 8 ohm speaker directly. To boost the power level of the loudspeakers to be used in a large room, the outputs of 8253A's are sent to a bank of power transisters (2N3055) in the emitter follower configuration. Using 5 V DC power supply on the collector side of the power transistors, one can obtain about 2 W audio output from each 8 ohm speaker. As the power transistors are turned fully on or fully off, the power to sound conversion is very efficient. The voices can fill a medium sized church without any problem.

I did not try to control the volume of the speakers, nor the wave shape. The speakers provide some filtering at either ends of the frequency spectrum. In the current configuration I have only used 6 speakers, which limits the number of voices to be played out.

An interesting behavior of this setup is that the 8253A can drive the speakers even with the 5 V power supply to the collectors of 2N3055 turned off. This way I can still play the organ while programming and debugging without generating too much noise to other family members. Apparently, 8253A puts out enough current through the base-emitter diode in 2N3055 to drive the speaker at the emitter end.

The Parallel Interface Card also has 4 8255 parallel I/O chips. I didn't use them so far. Since they give me 96 programmable digital I/O lines, I could use them to monitor a large piano keyboard as an alternate input device. I did not realize the advantage of this input route until I spent a long time key in all the music notes in the Toccata. It would be much more efficient if I had a regular piano keyboard and had a good pianist to key in the notes as he played on the keyboard. The ASCII terminal keyboard is not optimized to enter music notes, especially those in Bach's pieces.

3. THE CONTROL SOFTWARE

Programming is easy once the syntax is decided. The problem to encode polyphonic music can be eased greatly if we could design a good syntax to express the music. Here we will have to be able to include the following information in the code of a cord:

- The pitch and the channel number of every voice.
- The duration of every note in every voice.

A channel in 8253A can output a voice continuously after its frequency is set, until the frequency is changed or the channel is turned off explicitly. Therefore, when we want to play a new cord, we only have to specify the notes in voices which must be changed. Other unchanged voices do not have

to be manipulated explicitly. The cord will be sustained until the shortest note in the cord group ends, and a new cord must be specified to be played out next.

The syntax to specify a cord or a note group is as follows:

note1 note2 ... note'n mask duration

Up to 12 notes can be given before the mask. The mask word consists of up to 12 set bits which specify the active voices to be updated with the note values preceeding the mask. Bits and the corresponding channel numbers are shown in Fig. 29. A bit set in the mask must have a note value on the stack so that the note value can be deposit into the counter register of the specified channel.

Fig. 29. Bits in the Mask and the Active Channels

Mask Bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	chC	chB	chA	0	ch9	ch8	ch7	0	ch6	ch5	ch4	0	ch3	ch2	ch1
8253 #4				8253 #3				8253 #2				8253 #1			

Control Channels

Bits 3, 7, 11, and 15 should always be zero. The bit sequence actually corresponds to the 16 counter and control registers in the 8253's. Once these chips are initialized, the control registers should never be written into again. Hence bits 3, 7, 11, and 15 should always be zero to prevent writing into the control registers.

An example to change 6 channels of voices is as follows:

```
HEX   C2 G2 C3 E3 G3 C5   707  1/8
```

which assigns C2 to channel 9, G2 to channel 8, C3 to channel 7, E3 to channel 3, G3 to channel 2, and C5 to channel 1. The duration of this cord group is an 1/8 note. The note words, C2, ..., are defined as constants, leaving the counter valued on the stack. 707 is the mask, specifying 6 channels to be updated with 6 note values on the stack. 1/8 is the real actor which takes the mask and used the bit pattern in the mask to store note values into individual counter registers.

Using the above syntax, we can code any cord group, up to 12 voices. With a succession of cord groups, we can code measures, phrases, and complete pieces of polyphonic music.

The command set to realize this syntax structure is very simple and straightforward in Forth. There are basically only two screens of code as shown in Listing 15. The command PLAY refreshs a set of counter registers according to the mask n on the stack and sustain this cord group for a number of 20 milli-second waiting loops. This is the fundamental building block of the entire program. It allows us to play any combination of notes, up to 12 voices, through the speakers. All other commands are derived from PLAY.

The length of the 20 ms waiting loop can be adjusted by changing the contents in the variable FUDGE. For normal playing, a FUDGE factor of 25 is adequate. Allegro or fast pace requires a FUDGE factor of 10. For very slow movements, a FUDGE factor of 40 may be used. It has to be selected by trying to suit different music piece.

INITIATE writes to the control ports to configure all counter-timer channels operating in mode 2 to generate square wave. HUSH silences all 12 channels. HUSH is really cheating, because it is rather inconvenient to actually stop a channel from playing notes. What HUSH does is commanding all 12 channels to play at a frequency of 150 KHz which nobody can hear and no speaker can reproduce. It also tends to burn regular audio amplifiers which are sensitive to the high frequency voice.

BEAT defines all the cord playing commands, given names from 1/1, 1/2, 1/4, to 1/128. The names are chosen to indicate the duration of the cord group to be played. These names can be redefined to make code more readable.

8 octaves of notes are defined. The notes are simply constants of the counts to be deposit into the counter registers to produce the square waves of desired frequency. The notes are defined in Screens 4 to 8.

The note playing commands defined by BEAT in Screen 3 and the notes are all we need to encode polyphonic music pieces. The rest of the file ORGAN.BLK contains the code to play a few organ pieces by J. S. Bach. The pieces are coded by measures. Each measure is defined as a word. Many measures are combined to make up a phrase and phrases are combined to music piece. Coding each measure individually greatly eases the chores in debugging the piece. Each measure can be play as many times as needed and played at very slow speed to spot any discrepancy in notes and in timing.

Specialized tools are built whenever situation calls for them. Tools greatly simplifies coding, especially when the music pieces contain regular structures and patterns, as shown in Prelude in C major. When passages are repeated, the program can simply repeat measures already coded. However, since the syntax can only define cord groups in the vertical fashion,

it is not possible to transpose the repetitive themes occurring in different voices.

4. EXAMPLES

The code to perform Bach's Air On The G-String is shown in Screens 9 to 16. In Screen 9 there are some tools tailored to this piece. Let's first take a look at the first two measures of this music, as shown in Fig. 4 here. The most visible structure in this piece is the short, pulsing base with 1/16 notes separated by 1/16 rests. The melody is accompanied by 3 voice harmony. The tool word // thus takes four note values on the stack and play them out as a 1/16 cord through channel 7, 3, 2, and 1. The tool word //. is the same as //, except that after the cord is played for 1/16 note period, the base voice in channel 7 is silenced for 1/16 period of rest. //. is thus useful to code cords with 1/8 notes in the top three voices and the base is voiced only for 1/16 note period.

The word $ adds one 1/16 note to the base with a 1/16 rest. It is used to code passages where the top three voices are sustained while the base pulses.

The Air, as in most music pieces, contains many phrases which repeat. For example, the first two phrases P1 and P2 share the same 5 measures. As all the measures are defined as words they can be reused to save coding efforts. The phrase P3 is also repeated twice with some difference only in the last measure. The overall structure of this piece is very clear in the last definition of AIR, which plays this piece at a very slow pace with the FUDGE factor set to 60.

The C-Major Prelude in the Well Tempered Clavier is coded twice. The first effort is shown in Screens 17 to 21. The recurring structure of this piece is that each measure is composed of two repeating stretched cords. Each cord starts with the leading base and the tenor voice joins after 1/16 note period. Finally the soprano plays out a triple 1/16 note group. This structure is used to explore all the possible cord groups exhaustively and is closed by two 14 note cadenza.

In my first effort coding this Prelude, I defined two tool words to represent this cord structure. LEAD in Screen 18 plays the base and tenor leading voices, and CORD is Screen 17 plays the following soprano triple cord. The cord words Knn define the triple cords which can then be repeated in the half measure Mnn. In phrases Px, Mnn are all repeated twice to build full measures. This way, most of the repetition are taking care of.

Later, I felt that a much better job could be done, with the added feature that the repeating half measures are to be played by another group of three speakers, as I do have 6 speakers in my organ. This later effort is shown in Screens 22 to 26.

Since the leading base-tenor voices and the following soprano cord can be considered as one big 5 voice cord played out in sequence, only one tool word is needed to do a full measure. This tool word is defined as CORD in Screen 22. CORD first plays the 5 note cord group through channels 3, 2, and 1, and then repeats it through channels 9, 8, and 7. If we place the two groups of speakers at the opposing ends of a large hall, the echoing effect of these cords should be very interesting and impressive.

Using the new CORD, the entire piece can now be coded much more efficiently. The sub-definitions Knn and the repetitive half measured are all eliminated.

These are only two examples picked from a sizable library of Bach's music. My ambition is to build a complete machine readable music library of Bach's organ works. With the help of other Bach lovers, probably this can be achieved in a relatively short period of time.

5. CONCLUSION

I thoroughly enjoyed this project. I never could play any instrument because I couldn't imagine myself spending years learning to play one. Now I can proudly display my title as Organist/Programmer without being untruthful. The organ sounds very mechanical still. However, as far as Bach's music is concerned, a computer does very well because the tempo and the volume are generally kept constant. The interweaving of voices and the harmonic progressions are logical and almost mathematical. As the great master claimed: "It is very easy to play. You just have to hit the right keys at the right time, and the music will play itself!" This organ surely can hit right keys at the right time, and I trust that Bach will take care of the music by himself. By separating the voice spatially, polyphonic music is very attractive when played in very large room, providing feeling of depth more real than quadraphonic systems.

The method I developed here to encode complicated music pieces is a working solution to the problem of recording music in machine readable form. Forth allow us to build some super-structures on top of this system to simplify the coding effort somewhat, but I don't see much improvement further than that. There are many computer music systems available on various personal computers. Many have quite sophisticated coding methods for music encoding into machine readable form. However, when we used graphic methods to encode music, the resulting form are generally not transportable to other machines. This method I proposed is quite general because it uses only ASCII text. It may help professional musicians in exploring and exchange computerized music.

6. PARTS AND MUSIC AVAILABLE FROM OFFETE ENTERPRISES

MPIO-4 Parallel Interface Card for IBM PC $159.00
4 8253A-5's and 4 8255A-5's. 12 timer-counter channels and 96 programmable I/O lines. Suitable for stepper motor control and robotic applications, besides electronic organ.

Offete Organ Series, Vol. I. $15.00
IBM 5.25" disk. Source and object code of selected Two Part Invention, Preludes and Fugues, and Toccata in G Major.

Offete Organ Series, Vol. II. $15.00
IBM 5.25" disk. Source and object code of 24 selected pieces from Anna Magdalena' Notebook.

Offete Organ Series, Vol. III. $15.00
IBM 5.25" disk. Source and object code of Jesu, Joy of Man's Desiring, Sheep May Safely Graze, Christ Lay in Death's Prison, and In Deep Grief I Called Ye.

Offete Organ Series, Vol. IV. $15.00
IBM 5.25" disk. Source and object code of Fugue in G Minor and the Great Toccata and Fugue in D Minor.

Offete Organ Series, Vol. V. $15.00
IBM 5.25" disk. Source and object code of Prelude and Fugue in E Flat Major, the St. Anne's.

```
 0
 0   Demonstration of Electronic Organ with PC
 1
 2          Copyright C. H. Ting, 1986
 3
 4          Offete Enterprises, Inc.
 5          1306 South B Street
 6          San Mateo, CA  94402
 7          (415) 574-8250
 8
 9
10
11
12
13
14
15
```

```
 1
 0 \ AIR
 1 2 8 THRU ( Notes and cords )
 2 9 16 THRU ( Air on the 6-String )
 3 18 19 THRU ( Doxology )
 4 24 27 THRU ( Prelude in C Major, Version 1 )
 5 30 34 THRU ( Prelude in C major, Version 2 )
 6
 7
 8
 9
10
11
12
13
14
15
```

```
 2
 0 \ Muliple notes                                    04jan86cht
 1 25 FUDGE ! ( 1 mili-second)
 2 HEX 310 CONSTANT CHANNEL DECIMAL
 3 : (PLAY) ( period channel -)
 4    CHANNEL + >R   DUP R@ PC!   FLIP R> PC!   ;
 5 : PLAY ( .. n as -, send notes to channels specified by mask n)
 6    >R ( time)   16 0 DO   DUP 1 AND
 7       IF SWAP I (PLAY) THEN   2/   LOOP   DROP
 8    R>   0 DO 20 MS LOOP   KEY? ABORT" killed"   ;
 9
10
11
12
13
14
15
```

```
 3
02feb86cht \ Initiate 8253 and hush them                  04jan86cht
           HEX
           : CONTROL-PORTS ( n)
             320 313 DO   DUP I PC!   4 +LOOP DROP ;
           : INITIATE ( set up all ports to be mode 3)
             36 CONTROL-PORTS   76 CONTROL-PORTS   B6 CONTROL-PORTS ;
           : ! ( silence all channels with 300 KHz output)
             4 4 4   4 4 4   4 4 4   4 4 4   7777 1 PLAY   ;
           : HUSH   ! ;
           INITIATE HUSH   DECIMAL
           : BEAT   CONSTANT   DOES> @ PLAY ;
           128 BEAT 1/1    64 BEAT 1/2    32 BEAT 1/4   16 BEAT 1/8
             8 BEAT 1/16    4 BEAT 1/32    2 BEAT 1/64   1 BEAT 1/128
            96 BEAT 3/4    48 BEAT 3/8    24 BEAT 3/16  12 BEAT 3/32
           192 BEAT 3/2
```

```
 4
02feb86cht \ Notes                                        02feb86cht
           : NOTE ( n)   5950 ROT */  CONSTANT  ( DOES> @ PERIOD)  ;
           3270 10000 NOTE C1          3465 10000 NOTE C1#
           3671 10000 NOTE D1          3889 10000 NOTE D1#
           4120 10000 NOTE E1          4365 10000 NOTE F1
           4625 10000 NOTE F1#         4900 10000 NOTE G1
           5191 10000 NOTE G1#         5500 10000 NOTE A1
           5827 10000 NOTE A1#         6174 10000 NOTE B1
           6541 10000 NOTE C2          6930 10000 NOTE C2#
           7342 10000 NOTE D2          7778 10000 NOTE D2#
           8241 10000 NOTE E2          8731 10000 NOTE F2
           9250 10000 NOTE F2#         9800 10000 NOTE G2
           10383 10000 NOTE G2#        11000 10000 NOTE A2
           11654 10000 NOTE A2#        12347 10000 NOTE B2
```

```
 5
04jan86cht \ Notes                                        31dec85cht
           13081 10000 NOTE C3         13859 10000 NOTE C3#
           14683 10000 NOTE D3         15556 10000 NOTE D3#
           16481 10000 NOTE E3         17461 10000 NOTE F3
           18500 10000 NOTE F3#        19600 10000 NOTE G3
           20765 10000 NOTE G3#        22000 10000 NOTE A3
           23308 10000 NOTE A3#        24694 10000 NOTE B3

           2616 1000 NOTE C4           2772 1000 NOTE C4#
           2937 1000 NOTE D4           3111 1000 NOTE D4#
           3296 1000 NOTE E4           3492 1000 NOTE F4
           3700 1000 NOTE F4#          3920 1000 NOTE G4
           4153 1000 NOTE G4#          4400 1000 NOTE A4
           4662 1000 NOTE A4#          4939 1000 NOTE B4
```

Listing 15. Demonstration of electronic organ

```
      6
 0 \ Notes
 1 5233 1000 NOTE C5          5544 1000 NOTE C5#
 2 5873 1000 NOTE D5          6223 1000 NOTE D5#
 3 6593 1000 NOTE E5          6985 1000 NOTE F5
 4 7400 1000 NOTE F5#         7840 1000 NOTE G5
 5 8306 1000 NOTE G5#         8800 1000 NOTE A5
 6 9323 1000 NOTE A5#         9878 1000 NOTE B5
 7
 8 10465 1000 NOTE C6         11087 1000 NOTE C6#
 9 11747 1000 NOTE D6         12445 1000 NOTE D6#
10 13185 1000 NOTE E6         13969 1000 NOTE F6
11 14800 1000 NOTE F6#        15680 1000 NOTE G6
12 16612 1000 NOTE G6#        17600 1000 NOTE A6
13 18647 1000 NOTE A6#        19755 1000 NOTE B6
14
15
```

```
      7
 0 \ Notes
 1 2093 100 NOTE C7           2218 100 NOTE C7#
 2 2349 100 NOTE D7           2489 100 NOTE D7#
 3 2637 100 NOTE E7           2794 100 NOTE F7
 4 2960 100 NOTE F7#          3136 100 NOTE G7
 5 3322 100 NOTE G7#          3520 100 NOTE A7
 6 3729 100 NOTE A7#          3951 100 NOTE B7
 7
 8 4186 100 NOTE C8           4435 100 NOTE C8#
 9 4699 100 NOTE D8           4978 100 NOTE D8#
10 5274 100 NOTE E8           5588 100 NOTE F8
11 5920 100 NOTE F8#          6272 100 NOTE G8
12 6645 100 NOTE G8#          7040 100 NOTE A8
13 7459 100 NOTE A8#          7902 100 NOTE B8
14
15
```

```
      8
 0 \ Flat notes
 1 C1# CONSTANT D1b    D1# CONSTANT E1b    F1# CONSTANT G1b
 2 C2# CONSTANT D2b    D2# CONSTANT E2b    F2# CONSTANT G2b
 3 C3# CONSTANT D3b    D3# CONSTANT E3b    F3# CONSTANT G3b
 4 C4# CONSTANT D4b    D4# CONSTANT E4b    F4# CONSTANT G4b
 5 C5# CONSTANT D5b    D5# CONSTANT E5b    F5# CONSTANT G5b
 6 C6# CONSTANT D6b    D6# CONSTANT E6b    F6# CONSTANT G6b
 7 C7# CONSTANT D7b    D7# CONSTANT E7b    F7# CONSTANT G7b
 8 C8# CONSTANT D8b    D8# CONSTANT E8b    F8# CONSTANT G8b
 9 G1# CONSTANT A1b    A1# CONSTANT B1b
10 G2# CONSTANT A2b    A2# CONSTANT B2b
11 G3# CONSTANT A3b    A3# CONSTANT B3b
12 G4# CONSTANT A4b    A4# CONSTANT B4b
13 G5# CONSTANT A5b    A5# CONSTANT B5b
14 G6# CONSTANT A6b    A6# CONSTANT B6b
15 G7# CONSTANT A7b    A7# CONSTANT B7b
```

```
      9
31dec85cht \ Air for the G-String                    01feb86cht
           HEX
           : // ( n1 n2 n3 n4 )   701 1/16 ;
           : //. ( n1 n2 n3 n4 )   // 4 400 1/16 ;
           : $ ( n )   400 1/16  4 400 1/16 ;
```

```
     10
31dec85cht \ Air for the G-String                    01feb86cht
          : M1  C3 G4 C5 E4 //.  C4 $  B3 $  B2 $  A2 A4 C5 E4 //.
            A3 $ G3 $ G2 $ ;
          : M2  F2 A4 C5 E4 //.  F3 A4 401 1/16 4 F4 401 1/16
            F3# D4 A4 D4 701 1/16  4 C4 401 1/16  F2# B3 401 1/16
            4 C4 401 1/16  G2 D4 G4 B3 //.  G3 $ 4 400 1/32 A3 1 1/32
            F3 G3 401 1/16 4 400 1/16  F2 $ ! ;
          : M3  E2 E4 G4 G4 //.  E3 C4# B4b G4 //  4 A4 500 1/16
            D3 B4b 500 1/16 4 400 1/16  C2# E4 A4 G4 // 4 E4 401 1/16
            C3# 4 4 B3b // 4 A3 401 1/16  A2 D4 401 1/16 4 C4# 401 1/16
            A3 G4 401 1/16 4 F4 401 1/16 ! ;
          : M4  D2 D4 A4 F4 //.  D3 A4 D5 700 1/16 4 C5 500 1/16
            C3 D5 500 1/16 4 E5 500 1/16  C2 F5 500 1/16 4 D5 500 1/16
            B1 D4 G4 F4 // 4 D4 401 1/16  B2 4 4 A3 // G3 1 1/16
            G2 C4 401 1/16 4 B3 401 1/16  G3 F4 401 1/16 E4 1 1/16 ! ;
```

```
     11
31dec85cht \ Air for the G-String                    02feb86cht
          : M5  C3 C4 G4 E4 //.  C4 $ B3 $  B2 D4 G4 F4# // 4 G4 401 1/16
            A2 E4 G4 C4 //.  A3 C4 F4# C4 701 1/32 D4 1 1/32
            4 G4 E4 501 1/16  F3# A3 A4 E4 // 4 D4 401 1/16
            D3 D4 F4# D4 //  4 C4 401 1/16 ! ;
          : M6  G3 D4 G4 B3 // 4 A3 401 1/16 !  C3 E4 G4 A3 701 1/32
            B3 1 1/32 4 C4 401 1/16 !  D3 A3 G4 C4 //.
            D2 D4 F4# B3 // 4 A3 401 1/16 ;
          : M6  M6
            G2 G3 G4 G3 //  A2 400 1/16 B2 400 1/16 C3 400 1/16
            D3 400 1/16 F3 400 1/16 E3 400 1/16 D3 400 1/16 ! ;
          : M7  M6 G2 G3 G4 G3 701 3/8 !  4 1 1/8 ;
          : P1   M1 M2 M3 M4 M5 M6 ;
          : P2   M1 M2 M3 M4 M5 M7 ;
```

Listing 15. Demonstration of electronic organ (cont'd)

```
     12
 0 \ Air for the G-String                                  02feb86cht
 1 : M8  G2 D4 G4 B3 //.  G3 $ F3 400 1/16 4 C4 401 1/32
 2    B3 1 1/32 F2 A3 401 1/32 B3 1 1/32 4 G3 401 1/16
 3    E2 D4 G4 G4 // 4 A4 500 1/16 E3 C4# B4b G4 // 4 D4 600 1/16
 4    D3 E4 600 1/16 4 A4 500 1/16  D2 G4 B3b 501 1/16
 5    F4 100 1/16 ! ;
 6 : M9  C2# E4 E4 A3 // F4 F4 300 1/16  C3# G4 G4 A4 //
 7    4 E4 E4 700 1/16  E3 C4# C4# A4 // 4 G4 401 1/16
 8    A2 A4 C5# F4 // 4 E4 401 1/16 !
 9    D3 A4 D5 F4 //  D4 $ C4 A3 D4 F4 701 1/32 E4 1 1/32
10    4 D4 401 1/32 C4 1 1/32  C3 B3 401 1/16 4 A3 401 1/16 ! ;
11 : M10  B2 B3 D4 G3# // 4 C4 600 1/16  B3 D4 4 A3 //
12    4 E4 4 B3 // A3 F4 D5 B3 // 4 E4 D5 C4 // A2 F4 D5 D4 //
13    4 D4 600 1/16  G2# E4 D5 D4 // 4 C5 E4 501 1/16
14    A2 D4 B4 F4 // 4 C4 A4 700 1/16  B2 B3 G4# F4 //
15    4 A4 500 1/16 G2 E4 B4 E4 //. ! ;

     13
 0 \ Air for the G-String                                  01feb86cht
 1 : M11  A2 E4 A4 D4 // 4 C4 401 1/16 F3 D4 A4 B3 // 4 C4 A3
 2   601 1/16  D3 F4 A4 B3 // 4 C4 401 1/32 D4 1 1/32
 3    E3 E4 G4# C4 // 4 D4 B3 601 1/16 !  A2 C4 E4 A3 //.
 4    A3 $ G3 $ G2 $ ! ;
 5 : M12  F2# A3 D4 C4 //. F3# A4 600 1/16 4 400 1/16
 6    E3 E4 G4 C4 // 4 F4# E4 501 1/16 E2 E4 G4 D4 //
 7    C4 1 1/16 !  D2 A3 F4# A4 //.  D3 D4 600 1/16
 8    4 E4 E4 700 1/16  C3 D4 F4# A4 // 4 G4 500 1/16
 9    C2 D4 A4 G4 // 4 F4# 401 1/16 ! ;
10 : M13  B1 D4 A4 E4 701 1/32 D4 1 1/32  4 G4 401 1/16
11    B2 D4 G4 G3 //.  C3 E4 G4 A3 //.
12    D3 D4 F4# A3 // 4 B3 401 1/32 C4 1 1/32
13    G2 D4 G4 B3 //.  G3 400 1/16 4 C4 600 1/32 A4 1 1/32
14    F3 B3 G4 G3 // 4 C4 600 1/16  F2 D4 600 1/16 4 B3 600 1/16
15    ! ;

     14
 0 \ Air for the G-String                                  01feb86cht
 1 : M14  E2 G4 G4 C4 //.  E3 C4 A4 C4 // 4 B4b 500 1/16
 2    F3 A4 500 1/16 4 B4 500 1/16  F2 A4 C5 E4 // 4 D4 401 1/16
 3    ! F2# A4 C5 D4 //.  F3# D4 B4 D4 // 4 A4 500 1/16
 4    G3 B4 500 1/16 4 C5# 500 1/16  G2 B4 D5 F4 // E4 1 1/16 ! ;
 5 : M15  G2# B4 D5 E4 //.  G3# E4 C5# 700 1/16 4 B4 500 1/16
 6    A3 C5# 500 1/16 4 D5 500 1/16 A2 C5# E5 G4 // 4 F4 401 1/16 !
 7    D3 A4 E5 F4 // 4 C5# 500 1/16  D4 4 D5 700 1/16 4 A4 500 1/16
 8    C4 D4 500 1/16 4 A4 500 1/16  C3 F4 500 1/16 4 D4 500 1/16 ;
 9 : M16  B2 D4 G4 G3 // 4 B3 600 1/16  B3 D4 F4 700 1/16
10    4 G4 600 1/16  G3 E4 B4 G3 // 4 B3 401 1/16  B3 D4 G4 D4 //
11    4 F4 401 1/16  C4 C4 G4 F4 // 4 D4 401 1/16  C3 C4 B4 E4 //
12    4 C5 500 1/16  B2b C4 G4 E4 //.  B3b E4 401 1/16
13    4 F4 401 1/32 G4 1 1/32 ! ;
14
15

     15
 0 \ Air for the G-String                                  01feb86cht
 1 : M17  A3 C4 G4 C4 //.  A2 D4 F4 C4 //.  G2 E4 G4 C4 //
 2    4 E4 401 1/16  G3 G4 401 1/16 4 B4b 401 1/16 !
 3    F3 C4 F4 B4b //.  F2 A4 401 1/16 4 400 1/16 E2 $
 4    E3 C4 401 1/16 4 400 1/16 ! ;
 5 : M18  D3 D4 F4 B3 // 4 A3 D4 601 1/16  D2 D4 A4 F4 //
 6    4 F4 600 1/16  C2 A4 D5 F4 // 4 G4 600 1/16
 7    C3 F4 A3 601 1/16 4 C5 500 1/16  G2 G4 B4 D4 701 1/32
 8    E4 1 1/32  4 A4 F4 501 1/16  C3 G4 G4 F4 // 4 E4 401 1/16
 9    F3 F4 A4 700 1/16 D4 1 1/16 ! ;
10 : M19  G3 E4 G4 C4 701 1/32 B3 1 1/32 4 A3 401 1/16
11    F3 400 1/16 4 B3 401 1/16 ! G3 D4 F4 B3 // 4 E4 500 1/16
12    G2 G3 F4 B3 // 4 C4 401 1/16 !  C3 G3 E4 C4 //.
13    G2 $ E2 $ C2 $ ! ;

     16
 0 \ Air for the G-String                                  02feb86cht
 1 : M20  G2 G3 E4 G4 C4 703 1/32 B3 1 1/32 4 4 A4 601 1/16
 2    F2 F3 600 1/16 4 4 B3 601 1/16  G2 G3 D4 F4 B3 703 1/16
 3    4 4 600 1/32  A3 1 1/32  G1 G2 G3 B3 // C4 1 1/16
 4    C2 C3 G3 E4 701 1/1 ! ;
 5 : P3  M8 M9 M10 M11 M12 M13 M14 M15 M16 M17 M18 ;
 6 DECIMAL
 7 : AIR  60 FUDGE ! INITIATE  P1 P2 P3 M19  P3 M20 ;

     18
 0 \ Doxology                                              03jan86cht
 1 : M1  G3 B3 D4 G4 263 1/4 !  G3 B3 D4 G4 263 1/4
 2    D3 A3 D4 F4# 263 1/4  E3 G3 B3 E4 263 1/4
 3    B2 F3# B3 D4 263 1/4 !  E3 G3 B3 G4 263 1/4
 4    D3 F3# D4 A4 263 1/4  G2 G3 D4 B4 263 1/2 ! ;
 5 : M2  G3 G3 D4 B4 263 1/4 !  G3 B3 D3 B4 263 1/4
 6    G3 G3 D4 B4 263 1/4  D3 F3# D4 A4 263 1/4
 7    E3 G3 B3 G4 263 1/4 !  C3 G3 E4 C5 263 1/4
 8    G2 G3 D4 B4 263 1/4  D3 F3# D4 A4 263 1/2 ! ;
 9 : M3  G3 B3 D4 G4 263 1/4  F3# A3 D4 A4 263 1/4
10    G3 G3 D4 B4 263 1/4  D3 F3# D4 A4 263 1/8  C3 256 1/8
11    B2 G3 D4 G4 263 1/4 !  C3 G3 C4 E4 263 1/4
12    A2 A3 C4 F4# 263 1/4  G2 D3 B3 G4 263 1/2 ! ;
```

Listing 15. Demonstration of electronic organ (cont'd)

```
     19
 0 \ Doxology                                          03jan86cht
 1 : M4   G3 B3 G4 D5 263 1/4 ;   G3 G3 D4 B4 263 1/4
 2   E3 B3 E4 G4 263 1/4   D3 D4 F4# A4 263 1/4
 3  A2 C4 E4 C5 263 1/8 F4# 2 1/8   B2 D4 G4 B4 263 1/8 C3 256 1/8
 4   D3 D4 F4# A4 263 1/8 C4 4 1/8   G2 B3 G4 G4 263 1/2 ; ;
 5 : AMEN   C3 C4 E4 G4 263 1/2   G2 B3 D4 G4 263 1/2 ; ;
 6 : DOXOLOGY   M1 M2 M3 M4 4 1 1/4   AMEN ;
 7
 8
 9
10
11
12
13
14
15

     24
 0 \ Prelude 1, C Major                                02jan86cht
 1 : CORD ( n1 n2 n3)   SWAP ROT   3 0 DO   1 1/16   LOOP ;
 2 : K1   G4 C5 E5 CORD ;
 3 : M1   C4 4 4 7 1/16   E4 2 1/16   K1 K1 ; ;
 4 : K2   A4 D5 F5 CORD ;
 5 : M2   C4 4 4 7 1/16   D4 2 1/16   K2 K2 ; ;
 6 : K3   G4 D5 F5 CORD ;
 7 : M3   B3 4 4 7 1/16   D4 2 1/16   K3 K3 ; ;
 8 : K4   A4 E5 A5 CORD ;
 9 : M4   C4 4 4 7 1/16   E4 2 1/16   K4 K4 ; ;
10 : K5   F4# A4 D5 CORD ;
11 : M5   C4 4 4 7 1/16   D4 2 1/16   K5 K5 ; ;
12 : K6   G4 D5 G5 CORD ;
13 : M6   B3 4 4 7 1/16   D4 2 1/16   K6 K6 ; ;
14 : P1   M1 M1   M2 M2   M3 M3   M1 M1   M4 M4   M5 M5   M6 M6 ;
15

     25
 0 \ Prelude 1, C Major                                02jan86cht
 1 : LEAD ( n1 n2)   SWAP 4 4 7 1/16   2 1/16 ;
 2 : K7   E4 G4 C5 CORD ;
 3 : M7   B3 C4 LEAD   K7 K7 ; ;
 4 : M8   A3 C4 LEAD   K7 K7 ; ;
 5 : K9   D4 F4# C5 CORD ;
 6 : M9   D3 A3 LEAD   K9 K9 ; ;
 7 : K10   D4 G4 B4 CORD ;
 8 : M10   G3 B3 LEAD   K10 K10 ; ;
 9 : K11   E4 G4 C5# CORD ;
10 : M11   G3 A3# LEAD   K11 K11 ; ;
11 : K12   D4 A4 D5 CORD ;
12 : M12   F3 A3 LEAD   K12 K12 ; ;
13 : K13   D4 F4 B4 CORD ;
14 : M13   F3 G3# LEAD   K13 K13 ; ;
15 : P2   M7 M7 M8 M8 M9 M9 M10 M10 M11 M11 M12 M12 M13 M13 ;

     26
 0 \ Prelude 1, C Major                                02jan86cht
 1 : K14   C4 G4 C5 CORD ;       : M14   E3 G3 LEAD K14 K14 ; ;
 2 : K15   A3 C4 F4 CORD ;       : M15   E3 F3 LEAD K15 K15 ; ;
 3 : M16   D3 F3 LEAD   K15 K15 ; ;
 4 : K17   G3 B3 F4 CORD ;       : M17   G2 D3 LEAD K17 K17 ; ;
 5 : K18   G3 C4 E4 CORD ;       : M18   C3 E3 LEAD K18 K18 ; ;
 6 : K19   A3# C4 E4 CORD ;      : M19   C3 G3 LEAD K19 K19 ; ;
 7 : K20   A3 C4 E4 CORD ;       : M20   F2 F3 LEAD K20 K20 ; ;
 8 : K21   A3 C4 D4# CORD ;      : M21   F2# C3 LEAD K21 K21 ; ;
 9 : K22   B3 C4 D4 CORD ;       : M22   G2# F3 LEAD K22 K22 ; ;
10 : K23   G3 B3 D4 CORD ;       : M23   G2 F3 LEAD K23 K23 ; ;
11 : K24   K18 ;                 : M24   G2 E3 LEAD K18 K18 ; ;
12 : K25   G3 C4 F4 CORD ;       : M25   G2 D3 LEAD K25 K25 ; ;
13 : K26   G3 B3 F4 CORD ;       : M26   G2 D3 LEAD K26 K26 ; ;
14 : P3   M14 M14 M15 M15 M16 M16 M17 M17 M18 M18 M19 M19
15    M20 M20 M21 M21 M22 M22 M23 M23 M24 M24 M25 M25 M26 M26 ;

     27
 0 \ Prelude 1, C Major                                02jan86cht
 1 : K27   A3 C3 F3# CORD ;      : M27   G2 D3# LEAD K27 K27 ; ;
 2 : K28   G3 C4 G4 CORD ;       : M28   G2 E3 LEAD K28 K28 ; ;
 3 : K29   G3 C4 F4 CORD ;       : M29   G2 D2 LEAD K29 K29 ; ;
 4 : K30   G3 B3 F4 CORD ;       : M30   G2 D2 LEAD K30 K30 ; ;
 5 : K31   G3 A3# E4 CORD ;      : M31   C2 C3 LEAD K31 K31 ; ;
 6 : 14NOTES   14 0 DO 1 1/16 LOOP ;
 7 : K32   D3 F3 D3 F3 A3 F3 A3 C4 A3 C4 F4 C4 A3 F3 14NOTES ; ;
 8 : M32   C2 C3 LEAD K32 ;
 9 : K33   D4 E4 F4 D4 B4 G4 B4 D5 B4 D5 F5 D5 B4 G4 14NOTES ; ;
10 : M33   C2 B2 LEAD K33 ;
11 : FINALE C2 512 1/16 C3 256 1/16 E4 4 1/16 G4 2 1/16 C5 1 1/1 ;
12 : P4   M27 M27 M28 M28 M29 M29 M30 M30 M31 M31 M32 M33
13    FINALE HUSH ;
14 : PRELUDE-1   P1 P2 P3 P4 ;
15 32 FUDGE !

     30
 0 \ Prelude 1, C Major                                10jan86cht
 1 HEX
 2 : CORD ( n1..n5)
 3    4 PICK 4 1/16   3 PICK 2 1/16   2 0 DO   2 PICK 1 1/16
 4    OVER 1 1/16   DUP 1 1/16 LOOP ;
 5    4 PICK 400 1/16   3 PICK 200 1/16   2 0 DO   2 PICK 100 1/16
 6    OVER 100 1/16   DUP 100 1/16 LOOP   5 DROPS ; ;
 7 DECIMAL
 8
 9
10
11
12
13
14
15
```

Listing 15. Demonstration of electronic organ (cont'd)

```
      31
 0 \ Prelude 1, C Major
 1 : M1   C4 E4  G4 C5 E5 CORD ;
 2 : M2   C4 D4  A4 D5 F5 CORD ;
 3 : M3   B3 D4  G4 D5 F5 CORD ;
 4 : M4   C4 E4  A4 E5 A5 CORD ;
 5 : M5   C4 D4  F4# A4 D5 CORD ;
 6 : M6   B3 D4  G4 D5 G5 CORD ;
 7 : P1   M1 M2 M3 M1 M4 M5 M6 ;
 8
 9
10
11
12
13
14
15
```

```
      32
 0 \ Prelude 1, C Major
 1 : M7   B3 C4 E4 G4 C5 CORD ;
 2 : M8   A3 C4 E4 G4 C5 CORD ;
 3 : M9   D3 A3 D4 F4# C5 CORD ; ;
 4 : M10   G3 B3 D4 G4 B4 CORD ;
 5 : M11   G3 A3# E4 G4 C5# CORD ;
 6 : M12   F3 A3 D4 A4 D5 CORD ;
 7 : M13   F3 G3# D4 F4 B4 CORD ;
 8 : P2   M7 M8 M9 M10 M11 M12 M13 ;
 9
10
11
12
13
14
15
```

```
      33
 0 \ Prelude 1, C Major
 1 : M14   E3 G3 C4 G4 C5 CORD ;
 2 : M15   E3 F3 A3 C4 F4 CORD ;
 3 : M16   D3 F3 A3 C4 F4 CORD ;
 4 : M17   G2 D3 G3 B3 F4 CORD ;
 5 : M18   C3 E3 G3 C4 E4 CORD ;
 6 : M19   C3 G3 A3# C4 E4 CORD ;
 7 : M20   F2 F3 A3 C4 E4 CORD ;
 8 : M21   F2# C3 A3 C4 D4# CORD ;
 9 : M22   G2# F3 B3 C4 D4 CORD ;
10 : M23   G2 F3 G3 B3 D4 CORD ;
11 : M24   G2 E3 G3 C4 E4 CORD ;
12 : M25   G2 D3 G3 C4 F4 CORD ;
13 : M26   G2 D3 G3 B3 F4 CORD ;
14 : P3   M14 M15 M16 M17 M18 M19 M20 M21 M22 M23 M24 M25 M26 ;
15
```

```
      34
10jan86cht \ Prelude 1, C Major                                10jan86cht
          : M27   G2 D3# A3 C3 F3# CORD ;
          : M28   G2 E3 G3 C4 G4 CORD ;
          : M29   G2 D2 G3 C4 F4 CORD ;
          : M30   G2 D2 G3 B3 F4 CORD ;
          : M31   C2 C3 G3 A3# E4 CORD ;
          : 14NOTES   14 0 DO 1 1/16 LOOP ;
          : K32   D3 F3 D3 F3 A3 F3 A3 C4 A3 C4 F4 C4 A3 F3 14NOTES ! ;
          : M32   C2 4 1/16 C3 2 1/16 K32 ;
          : K33   D4 E4 F4 D4 B4 G4 B4 D5 B4 D5 F5 D5 B4 G4  ;  HEX
          : M33   C2 400 1/16 B2 200 1/16 K33 0E 0 DO 100 1/16 LOOP ! ;
          : FINALE C2 400 1/16 C3 100 1/16 E4 4 1/16 G4 2 1/16 C5 1 1/1 ;
          : P4   M27 M28 M29 M30 M31 M32 M33 FINALE ! ;
          : PRELUDE-1'   P1 P2 P3 P4 ;
          DECIMAL   32 FUDGE !
```

```
                                                                0
10jan86cht      Demonstration of Electronic Organ with PC       02feb86cht

                     Copyright C. H. Ting, 1986

                     Offete Enterprises, Inc.
                     1306 South B Street
                     San Mateo, CA  94402
                     (415) 574-8250
```

```
                                                                0
10jan86cht      Demonstration of Electronic Organ with PC       02feb86cht

                     Copyright C. H. Ting, 1986

                     Offete Enterprises, Inc.
                     1306 South B Street
                     San Mateo, CA  94402
                     (415) 574-8250
```

Listing 15. Demonstration of electronic organ (cont'd)

XV. PROGRAMME OF AN ORGAN RECITAL

BACH ORGAN RECITAL

On a Six Channel Computer organ
Dr. C. H. Ting
Sunday, April 12, 1987, 7:00 pm.
St. Andrew's Lutheran Church, San Mateo, CA

Prelude and Fugue in C Major, BWV 846

Prelude and Fugue in C Minor, BWV 847

In Deep Grief I Cry to Thee, BWV 686

Chorale Preludes Adapted to Organ by E. Power Bigg

Christ Lay in Death's Boundage, Cantata No. 4

Jesu, Joy of Man's Desiring, Cantata No. 147

Sheep May Safely Graze, Cantata No. 208

Intermission

Toccata, Adagio and Allegro in G Major, BWV 916

Fugue in G Minor, BWV 578

Toccata and Fugue in D Minor, BWV 565

Prelude and Fugue in E Flat Major, BWV 552

Sponsored by the Christ Church of the Bay Area, San Mateo

The Computer Organ

This computer organ was conceived, built and programmed by Dr. C. H. Ting to experiment music making using a personal computer. His goal was a simple, low cost, and self contained system which can be used to enter music score and play back his favorite organ music by J. S. Bach. Inspite of the simplicity of this computer organ, the tonal quality is remarkably similar to an early pipe organ. With six channels of output, the organ is capable of reproducing all the organ pieces written by Bach and the Baroque masters.

The most interesting feature of this computer organ is that different voices in a contrapuntal composition can be assigned to different speakers which can be separated by long distance in a large hall. It is thus possible to 'spatially separate' the voices in a contrapuntal music and fill the hall with voices coming to the listeners from many different directions, creating a truely three dimensions effect of the music. This is another method to help clarify Bach's organ music and to aid its perception by the listeners.

To make a computer intelligent enough to play music of Bach's sophistication, special programming tools must be built so that the music score can be converted to a form which is both human readable and machine readable. The computer must be able to read it so that it can execute or play the music. Human must also be able to read it so that he can edit and control the music for proper presentation. Dr. Ting developed a music description language by which he coded a large collection of Bach's organ works, including the ones presented in this recital. This language consists of rules to construct sequences of chords and to string them together to form a playable piece. Each chord is specified by the notes in it, the assignments of notes to channels, and the duration of the chord. Measures are thus defined in terms of chord progressions, phrases in terms of measures, and whole pieces in terms of phrases.

This music description language is based upon the very powerful programming language 'FORTH' which has been widely used for machine control and automated instrumentation. Many attributes of FORTH, such as fast execution speed, efficient memory utilization, ease in high level construction, and interactive interface between user and computer, make it possible to encode large and complicated music for the computer to process and to play back.

The Music

The Preludes and Fugues in C Major and C Minor from the Well-Tempered Clavier, Book I, are the standard practice pieces for serious students of Bach. These were also the first ones coded after the computer organ was put together. They are paricularly interesting here because the chord progressings in the preludes are always repeated. The most plausible assumption is that the repeating chords are meant to be echoes of the leading chords. This echoing effect is best demonstrated by this computer organ. The six speakers are programmed so that two groups of three speakers are engaging in a dialog, projecting chords at each other across the hall.

The fugal prelude 'In Deep Grief I Cry to Thee' is a six part fugue using double padele. This is one piece in which all the six channels in the computer organ are gainfully occupied. The double padele effectively conveys the notion of deep grief in Martin Luther's original chorale. However, the joyous rhythum towards the end reassures the grace and salvation from above.

The chorale preludes were adapted from Bach's cantatas by E. Power Bigg to organ. Three movements, Sinfonia, Chorale and Variations, are presented in 'Christ Lay in Death's Boundage', from the Easter Cantata. The Sinfonia sets the mode with its slow and grave motive in the padele, sinking ever deeper into death's strong hold. After the declaration of Hallelujah at the end of the Chorale, the high voice in the Variations rises higher and higher, as if following the resurrected Christ towards his throne.

'Jesu, Joy of Man's Desiring' from Cantata No. 147 is probably Bach's most often played chorale prelude. The lively high voice in triplets hugs warmly around Schop's chorale melody of 1642, in the hymn of Martin Jahn's: "Jesus remains all my joy, my heart's solace and strength."

'Sheep May Safely Graze' is a chorale prelude from Cantata No. 208 for the birthday of Duke Christian of Sachzen-Weissenfels in 1716. The fresh, pastoral character of this music is a remarkable instance of Bach's power of tone painting. The pattern set in the first measures dipicts lambkins playfully jumping around their mothers who are engaged in the more serious bussiness of grazing.

The second half of this recital covers Bach from his youth to the last years in Leipzig. Toccata in G Major is a keyboard composition in his early Weimar period when he was liberating himself from the German masters through Italian influences. This piece was chosen here mainly because of the very powerfull decending chords in the Toccata. These chords require all six speakers sounding together, buildinga high contrast against the solo voices in the concerti style.

The 'Little' Fugue in G Minor and the 'Great' Toccata and Fugue were examples of Bach's maturity in his late Weimar years. The theme in the Fugue in G Minor was laid out so vigorously and with such rapid and weighty developments that it is impossible to find its parallel in Bach's contemporaries. As for the Toccata and Fugue in D Minor, we have the words from Albert Schweitzer: "The strong and ardent spirit has finally realized the laws of form. A single dramatic ground-thought unites the daring passage work of the toccata, that seems to pile up like wave on wave; and in the fugue the intercalated passage in broken chords only serve to make the climax all the more powerful".

The Prelude and Fugue in E Flat Major belonged to Bach's late Leipzig period when he tried to summerize his life's work. The theme in the fugue is from the ever popular hymn: "O God, our help in ages past, our hopes for years to come". The same theme recurs in three connected fugues, symbolizing the Trinity. The first fugue is calm and majestic, with an absolutely uniform movement throughout. In the second fugue, the theme seems to be disguised, and is only occasionally recognizable in its true shape, as if to suggest the devine assumption of an earthly form. In the third fugue, it is transformed into rushing semiquavers, as if the Pentecostal wind were coming roaring from heaven.

The Organist/Programmer

Dr. C. H. Ting is a chemist by training and a computer specialist by inclination. He graduated from the National Taiwan University in 1961 and obtained his Ph. D. in physical chemistry in 1965 at the University of Chicago. After two years of post doctorate research in US and Germany, he returned to Taiwan to teach chemistry and to carry on research in molecular spectroscopy, quantum chemistry, and chemical instrumentation. He returned to US in 1975 to pursue his new interests in electronics and computers, and joint the Lockheed Palo Alto Research Laboratories in 1976. Currently he is a staff scientist working on a number of projects concerning fast signal processing, image processing, instrumental control, and most recently large parallel processing systems.

He 'discovered' FORTH and used it as a software tool to develop image processing systems and was attracted to the power and effectiveness of this programming language. He joint the Forth Interest Group in 1979 and has been an active member in the Forth community, published many paper and books on FORTH. Forth Interest Group nominated him to the 'Figgy' award in 1986 for his contribution to this programming language. Besides using FORTH at work to solve serious problems, he brings it back home for recreational computing. The computer organ is one of his home projects which yielded audible results.

His musical experience was limited to the four years in Chicago, where he was housed with a music major, Mr. David Claypool, who later completed a Ph. D. degree in musicology at Northwestern University. Mr. Claypool was a feverish Bach partisan and left the same fever on Dr. Ting. The other music exposure was his occcassional duty to supervise the piano practice seesions of his three children. However, lacking serious musical training did not divert him from the determination that he can train his computer to play Bach faster than his children.

BACH ORGAN RECITAL

on a six channel computer organ

Dr. C. H. Ting
Sunday, April 12, 7 p.m.
St. Andrew's Lutheran Church
1501 South El Camino Real
San Mateo, California

Preludes
Fugues
Toccatas
Cantata Themes

Bach's organ pieces were translated by Dr. Ting into a form readable by an IBM Personal Computer. The organ-like sound is generated by the computer through a six channel tone generator and six speakers. Dr. Ting uses this system to experiment on the spatial separation of voices in Bach's music.

$10 Donation ($5 seniors and students)
towards the Building Fund for the Christ Church of the Bay Area.
For information call (415) 871-7770 or (415) 692-7842

OLD BACH PLAYS IBM-PC

A Computer Organ Recital
Dr. C. H. Ting
Sunday, April 12, 7 p.m.
St. Andrew's Lutheran Church

Bach's organ pieces were translated by Dr. Ting into a form readable by an IBM Personal Computer. The organ-like sound is generated by the computer through a six channel tone generator and six speakers. Dr. Ting uses this system to experiment on the spatial separation of voices in Bach's music.

$10 Donation ($5 seniors and students)
towards the Building Fund for theChrist Church of the Bay Area.
For information call (415) 871-7770 or (415) 692-7842

XV. A DEVIL'S DICTIONARY

1. INTRODUCTION

'The Devil's DP Dictionary' by Stan Kelly-Bootle(1) lists common terms used by programmers and interprets them in a humorous way. I thought that if I could put these terms in my computer and set up a convenient method to recall them, I would have an interesting toy with which to entertain my friends and guests. It is appropriate to implement this dictionary using Forth because Forth has all the necessary tools to build a dictionary. As a matter of fact, all the commands in Forth are contained in a dictionary, and the text interpreter in Forth is programmed to search this dictionary to identify commands typed by the user. With these important tolls already in place, I thought that it would be a simple matter to build the Devil's Dictionary on top of the Forth dictionary. The only other thing necessary is to be sure to secure permission from the publisher of the book.

The technique used to implement this dictionary should be of interest to many people because dictionary is a fundamental vehicle of information and intelligence. Consigning dictionaries to electronic storage devices will certainly make large amount of information readily available to users, who can then retrieve and use the information from such sources efficiently. In educational settings, an electronic dictionary can be useful to students, who can interact with the computer as they learn about subjects without having to go through rigidly scheduled classes. It is my hope that readers will take the implementation as a model and build dictionary systems that represent their own interests and applications.

2. THE ALMIGHTY \ COMMAND

To start this discussion, we can use the regular high level defining word : to define dictionary entries. A good example of this use is:

```
:  ABACUS  ." A reliable solid state device recently superceded
           by Cray I."  ;
```

After we define ABACUS as above, if we type ABACUS on the terminal, following it with a carriage return, the message between ." and " will be typed out on the CRT screen. This sequence of actions represents the idea in building a dictionary. An

entry has a name and an associated string in the dictionary. When we invoke the name, our system responds with the string.

A problem with this approach, however, is that the string will be compiled into RAM along with the entry name. Using a personal computer with a limited amount of RAM memory, we will not be able to build a dictionary of a useful size. We will do better if we only compile the names of entries with the minimum amount of information that will enable the computer to retrieve the string from disk when we invoke the name. Another consideration is how to store the dictionary entries on disk in the most convenient and efficient way.

These requirements call for the creation of a special defining command that can compile the dictionary entries from disk and also retrieve the associated string when we execute the names of entries. This defining command has the name \ in Screen 22, Listing 16. A defining command is a specialized Forth compiler-interpreter(2). The portion between CREATE and DOES> is the compiler which will be used to compile new commands into the Forth dictionary. The portion between DOES> and ; is the interpreter, which specifies how the new commands built by the defining command are executed when they are invoked.

An interesting property of this command is that it serves to purposes: it acts as a defining command and also as a delimiter to the string of the previous string. To understand why \ has to behave this way, let us first look at a block of text in which a few dictionary entries are defined, as shown in Screen 60 of the listing.

In Screen 60, the command \ is used syntactically as a separator between dictionary entries. Each entry comprises a name and a few sentences of explanation. The name and the string are separated by a space. However, the string is of variable length and must be allowed to have many spaces within it; the string is terminated by the \ command. It is, of course, possilbe to designate a special character other than \ as the string terminator, but that substitution would prove messy and inelegant.

What \ does, as shown in Screen 22, is first compile the entry name into the Forth dictionary as the name of a new Forth command, with three numbers in the parameter field: the block number where the associated string is stored, the character offset of the string from the beginning of the block, and the length of the string. Using these three parameter values, the defined dictionary entry will be able to locate its string on disk, retrieve it, and print it out on the CRT terminal. These activities are defined in execution time between DOES> and ; .

When a new entry is compiled, the actual content of the string is immaterial. All we need are the starting and ending locations of the string. The compiler scans the string until it detects the \ character. The scanning character pointer >IN then must be backed up by one character so that the \ character can be executed again to compile the next dictionary entry.

The creation of \ command allows us to use a simple syntax to store the texts of dictionary entries on disk in compact form. We can then use those texts to build the Devil's Dictionary, with entry names compiled into RAM while the text strings remain on disk, only to be retrieved in run time. This way, the RAM can be used efficiently. The system can search and identify dictionary entries quickly because those entries are in the RAM; the associated string will be printed out from the disk. This seems to be the best compromise among RAM usage, execution speed, and a large volume of text stored on disk.

3. OUTPUT OF DICTIONARY ENTRIES

Strings to be printed on the CRT terminal are of variable length, usually longer than one line. It is necessary to wrap words at the end of line to the next line without breaking them in the middle. Every word processor knows how to do this. The command DISPLAY (defined in Screen 21) does this word wrapping. It behaves similar to TYPE; however, a carriage return is inserted before a word that crosses the 64-character line boundary. The command -CHAR is similar to -TRAILING, except that it eliminates the trailing nonblank characters. ADJUST extracts a substring of fewer than 64 characters to be typed out by TYPE, with the location and length of the remaining string still on the stack.

In the execution part of the command \ , DISPLAY is used to type out the string associated with a dictionary entry so that the printout looks professional. The QUIT command after DISPLAY is used to suppress the "ok" message after successful execution of an entry. If the dictionary is to be used by people unfamiliar with Forth or any other computer language, it is better to leave these language prompts out of the CRT display.

4. THE FUZZY SEARCH

Computers are generally unfriendly because they expect the user to type in commands in the exact forms specified by the language in which the computer is programmed. Misspelling a single character in a command causes the computer to spit out some message that is incomprehensible to mortal beings. To help the user, we want the computer to search through its dictionary for entries similar to the command entered by the user when the command does not match exactly any one of the dictionary entry. The computer should then display the names of these entries and politely ask the user to try one of them.

The method we use to locate similar names in the dictionary is based on the Soundex Code scheme(3), which converts similarly pronounced words into one code. The Soundex code of a word starts with the first letter of the word, ignores all vowels, and groups the consonants into six groups. Only the first

three nonrepeating consonant codes are retained. If less than 3 codes are generated, it is zero filled to three codes. A word is thus converted to a code which contains one letter and three numbers. This system uses the Soundex code to search the dictionary. The names of entries with the same Soundex code will be printed on the CRT as suggestions for the user to try.

Screen 23 shows the Soundex code table and the command CHAR, which converts an alphabetic character to its corresponding Soundex code by looking up the code in the table with 26 letters. SDX is a buffer that stores the Soundex code generated from the word entered by the user; that code is later compared with codes derived from dictionary entries.

The command $CONVERT shown in Screen 24 converts a string into its Soundex code. Given the address of a string on the stack, $CONVERT will generate the corresponding Soundex code in the PAD buffer. $MOVE converts one letter into its Soundex equivalent and adds this digit to the Soundex code string in PAD buffer. However, $MOVE will not move a digit if it is 0 or if it repeats the previous digit--according to the Soundex conversion rule. ST is the variable storing the lastly converted digit; it allows us the keep track of the repeating digits.

In Screen 25, the command $GET converts one string into its Soundex code and copies the code from PAD buffer to SDX buffer. .ID prints the name of a dictionary entry. given its name field address on the stack. HASH is a special F83 command to select a vocabulary thread among four threads to search for a dictionary entry(4). In F83, every vocabulary in the Forth dictionary is hashed into four threads to accelerate the compiler and the text interpreter--only a quarter of the vocabulary is searched for any command.

.NAME in Screen 26 prints the name of a dictionary entry if its Soundex code agrees with the code stored in the SDX buffer. .NAMES will search through the dictionary and print out all the names in the dictionary having the same Soundex code, given the link field address of the entry used to start the search. Finally, the command SEARCH converts the user-entered word into a Soundex code, searches through the dictionary, and prints all entries of the same Soundex code.

With a figForth or 79-Standard Forth system, which do not hash the dictionary, one should eliminate the word HASH in the phrase CONTEXT @ HASH @ in the definition of SEARCH. The purpose of this phrase is to find the beginning of a thread to start the dictionary search.

5. THE INTERPRETER LOOP

We can write a new text interpreter(5) which will ask the user to enter a word, search the dictionary for this word, and print

the associated string if the word is in the dictionary. If the entered word is not an entry in the dictionary, SEARCH will be invoked to do the fuzzy search and print out all the names of dictionary entries with the same Soundex code. This is precisely what the Forth text interpreter does, with only one exception: when a word is not in the dictionary, SEARCH is called instead of NUMBER. In this system, after the new dictionary is compiled, there is not need for the regular Forth interpreter. We can replace the execution address of NUMBER in the text interpreter loop by the execution address of SEARCH. This is the cheapest way to implement our special purpose interpreter.

In F83, NUMBER is a deferred word, which executes the command whose execution address is placed in its parameter field. To change the behavior of NUMBER to SEARCH, we only have to deposit the execution address of SEARCH into the parameter field of NUMBER. In a system like fig-Forth, one will have to 'hot-patch' the address of NUMBER in INTERPRET loop and replace it with the address of SEARCH. This is a surgery not recommended in textbooks, but it is fun. However, do it carefully. The Forth system may 'bleed to death' if you cut into it at the wrong place. The formal way to do this is to rewrite INTERPRET with SEARCH and put the new INTERPRET in an infinite loop such as QUIT.

6. A SEAL VOCABULARY

One more problem we face is the partition of the Devil's Dictionary from the regular Forth dictionary. (The user should not be permitted to execute any regular Forth command that would mess up the system.) The best way to handle this task is to create a separate vocabulary to hold all the dictionary entries and seal off the other Forth vocabularies so that the user can only access the Devil's Dictionary. Different Forth systems provide different ways of building a seal vocabulary. In F83, a seal vocabulary can be constructed following the command sequence shown in Screen 20.

First, all the tools are compiled into the regular FORTH vocabulary, including the definition of a new vocabulary DEVIL in Screen 22.

Next, all the Devil's Dictionary entries are compiled into the DEVIL vocabulary. At this point the NUMBER command in INTERPRET is changed to SEARCH by the phrase

```
' SEARCH IS NUMBER
```

so as to activate the fuzzy search if an entered word failed to match a dictionary entry.

Lastly, the phrase SEAL DEVIL at the end of Screen 20 makes DEVIL the only vocabulary to be searched

by the text interpreter, thus sealing off other vocabularies.

7. THE LAST IMPROVEMENT

This Devil's Dictionary, once activated, loops on forever, searching the dictionary for every command the user types on the terminal. It prints Kelly-Bootle's interpretation of the word if that word is in the dictionary, or it prints a list of closely related words to help the user to continue the dialogue. Even though several hundreds words are defined in this dictionary, the fuzzy search very often fails to find any word of the same Soundex code as the entered word, and an empty list is returned. An empty list is not a satisfactory answer, and is annoying to the user.

The computer should always give the user some valid suggestions to carry on the conversation. In case the fuzzy search returns an empty list, the system will print all the entry names in a screen, as suggestions for the user to try. The screen number is stored in the variable SCR, which is incremented every time a screen is accessed.

A convenient way to print the names of dictionary entries in a screen is to redefine the command \ , as shown in Screen 27. It must be done after all the dictionary entries are compiled and before the DEVIL vocabulary is sealed. The new \ command will print one entry name and skip over the the explanation string. If the block containing many entries is loaded by the command n LOAD, all the entry names in screen n will be printed. SEARCH is this redefined to detect the condition of empty list and executes a SCR @ LOAD command to print names in a screen. The detailed command sequence to effect this improvement is shown in Screens 27 and 28.

Words in the Devil's Dictionary is shown in Figure 30. A sample dialogue with this Devil's Dictionary is shown in Figure 31. The computer tries to be helpful whether a word can be found or not.

8. CONCLUSION

I hope that I have made the technique of building an on-line electronic dictionary clear enough so that those readers with a working Forth system (F83 preferred) can implement it for fields involving their interests and expertise. This is a rather advanced Forth application, because the use of CREATE to compile dictionary entries. It also tinker with the text interpreter in the F83 system. Nevertheless, it is not too complicated to be incomprehensible, while still vividly demonstrates the power contained in Forth to implement special language syntax.

REFERENCES

1. Kelly-Bootle, Stan, 'The Devil's DP Dictionary', McGraw-Hill, Inc., 1981.

2. Harris, Kim R., 'Forth Extensibility', Byte, August 1980, pp. 164-184.

3. Jacobs, J. R., 'Finding Words That Sound Alike', Byte, March 1982, pp 473-474.

4. Perry, Michael, 'Vocabulary Mechanisms in Forth', FORML Conference Proceedings, 1980, pp. 39-41.

5. Ting, C. H., Systems Guide to figForth, Offete Enterprises, 1980, pp. 39-48.

WORDS
\ YOUR-PROGRAM YODALS XDS WORST-CASE WORD-PROCESSING WOM
WILD-CARD WHEEL WATERGATE VULNERABILITY VUE VTSO
VOLTAIRE-CANDIDE VMOS VIRTUAL LATEST-VERSION VERSION
VERIFICATION UDV UTM USER URN UPTIME UPGRADE UP
UNDETECTED UNDECIDABILITY UNBUNDLING UN- TWO'S-COMPLEMENT
TURNKEY TURNAROUND TURING-MACHINE TURING TTY TRUNCATE
TRIVIAL TRAVELING-SALESPERSON-PROBLEM TRANSPARENT TRAILER TPD
TOP-DOWN TM TIME-SLICE TIMESHARING TIME-MANAGEMENT TIME
THROWAWAY-CHARACTER THRASHING THINK TEXT-EDITOR
TERMINAL-DESEASES TEMPLATE TACKY-MAT SYSTEMS SYSTEM
SYMPOSIUM SWEETSHOP SUSPECT SUPERSTITION SUPERCOMPUTER
SUMMATION-CONVENTION SUBROUTINE STRUCTURED STRUCTURE STRINGENT
STEPWISE-REFINEMENT STATE-OF-THE-ART STANDBY STANDARD-DEVIATION
STACK STABILITY SSR SPOOL SPECTRUM SOURCE-CODE SOS
SOFTWARE-ROT SOFTWARE SNA SIZING SINGLE-CASE SIMPLEX
SIDEGRADE SHILOP SHIFT SEVEN-CATASTROPHES SERIAL
SEQUENTIAL-FILE SENIOR-SYSTEM-ANALYST SEMICONDUCTOR SCROLLING
SAWTEETH ST.-PRESPER REVERSED-CLASS-ACTION RESTROOM
RESPONSE-TIME REPORTAGE RELOAD RELEASE REFERENCE-ACCOUNT
REENTRANT REDUNDANCY RECURSIVE REAL-WORLD REALITY RDCM
RANDOM-FILE RANDOM QUEUE QUERY-PROGRAM QLP PUNCH PTF
PROSPECT PROPIETARY-CAVEAT PROPOSAL-EVALUATION PROPOSAL
PROCTOLOGIST PRIME-RATE PRICE/PERFORMANCE PRESTIDIGIT
PREFIX-NOTATION PRECEDENCE PRAYER POM POLISH-NOTATION
POACHING PLOTTER PLATEN PIGEONHOLE PHASE
PESSIMIZING-COMPILER PERSON-HOUR PERSONAL-COMPUTING PEER-GROUP
PAYROLL PAUSE PATCH PASSWORD PASCAL-MANUAL PARITY
PARENTHESIS PARENTHESES PARALLEL PAPER-LOW PAGING
PACKAGE-SWITCHING OXYMORON OVERHEAD OVERFLOW OSOPHOBIA OS
OR OR OPERATOR OPERATING-SYSTEM OPEN ONE-LINE-PATCH OGHAM
OGAM OFFICE-USE-ONLY OEM-COGS OEM OEDIPOS-COMPLEX OCR-B
OCR-A OCR OBSOLESCENCE OBJECT NUMEROLOGY NUMEROLATRY
NUMBER-CRUNCHER NULL NOT- NOR NON- NOISE
NOBEL-PRICE-WINNERS NIH NETWORK NETWOK NEST NATURALLER
NATURAL-LANGUAGE-COMPILER NATURAL-LANGUAGE NAND NACK
MY-PROGRAM MUSE MURPHY'S-LAW MUM MULTITASKING
MULTIPROCESSING MULTIPLEX MULTIJOBBING MULTI MTTR MTBF MSR
MOZ-DONG MOUNT MONTE-CARLO-METHOD MONOLITHIC MODULE MODULAR
MNEMONIC MIPS MINI-STRING MIDDLEWARE MIDDLE-OUT
MICROPROCESSOR MICRO MICR METHODOLOGY METAPROGRAMMER
MENDACITY-SEQUENCE MBT MAP MANUFACTURER MANIAC MAN-HOUR
MAJOR-NEW-LEVEL-RELEASE MAINTAINENCE MAFIA MACHINE-INDEPENDENT
LUDDITE LP LOW-LEVEL-LANGUAGE LORD-HIGH-FIXER LOOPHOLE
LOOP-ENDLESS LOOP LOGOMACHY LOGICAL-DIAGRAM LOCAL
LINEAR-PROGRAMMING LIFO LIBERATION LEXICON < LAMMA-THREE
LEADING-EDGE LEADER LAW LATEST-VERSION LABOR-HOUR LABEL
KSAM-FILE KLUDGE KING K JOHN-BIRCH-MACHINE JOB-TRICKLE JCL
JARGON-FILE JANUS ISO ISAM-FILE IRS IOU IO INTROSPECT
INTERFACE INSINNUENDO IN-HOUSE INCOMPLETENESS-THEOREM
INCH-WORM IMPLEMENTATION IMPERSONAL-COMPUTING IMP
IDEAL-BUSINESS-MACHINES ICARUS IBM HOWEVER HOLE HIRSUTE
HIGH-LEVEL-LANGUAGE HEXADECIMAL HEURISTICS HARTREE-CONSTANT
HARDWARE HARD-SECTOR HALTING-PROBLEM HAIRY HAIR GRUNGE

Figure 30. Words in the Devil's dictionary

```
GROSCH'S-LAW,COROLLARY-TO   GROSCH'S-LAW   GRITCH   GRAPHICS
GRANDFATHER   GOTO-ORDER   GOTO   GOSUB   GOLFBALL   GODOT   GOD
GLOBAL   GLITCH   GLASS-TTY   GLIDING-THE-LILY   GIGO   GERSHWIN'S-LAW
GENERAL-PURPOSE-GRAPHS   GEE-WHIZ   GATE   GANGPUNCH   FUNCTION   FS
FREELANCE   FORTRAN   FOOT-WORM   FOOLPROOF   FLOWCHART   FLOPPY-DRUM
FLOPPETTE   FLIP-FLOP   FLEEP   FIX   FIRST-TIME   FIRMWARE
FINITE-STATE   FIFO   FASTRAND   EXTENDED-BASIC   EXPER   EXIT
EXCEPTION-REPORTING   EWOM   ETHELRED-OS   EPSS   ENVIRONMENT
ENGLISH   END-USER   ENDLESS-LOOP   EMULATION   ELSE-DANGLING
ELECTRON   EDITOR   EBCDIC   E138   DYXLESIA   DYSLEXIA   DYNAMIC-HALT
DYNAMIC   DUMP   DP-VOGUE   DMP   DP-LITIGATION   DP-FRAUD
DP-DICTIONARY   DP-ATTORNEY   DOWNTIME   DOWN   DOUBLE-SIDED-DRUM
DOCUMENTATION   DISMAL   DEPILATION   DENIER   DELAY   DEGRADE
DEFAULT   DECOMPILER   DECISION-TABLE   DECADE-COUNTER   DEBUGGING
DEBUGGER   DEADLINE   DBMS   DATABASE-MANAGEMENT-SYSTEM   DATA-BANK
DATA   DANGLING-ELSE   DAISY-CHAIN   CURTATION   CURSOR-ADDRESS
CURSOR   CRT   CREED   CRASH   CPU   CPM   CORRECTRICE   CONVERSION
CONVENTION   CONSULTANT   CONSOLE   CONJECTURE   CONGRESS   COMPUTIBLE
COMPUTER-SCIENCE   COMPUTER-MUSIC   COMPUTER-JOURNALIST   COMPUTABLE
COMPLEX   COMPATIBLE   COMPATABLE   COMMON-LANGUAGE   COME-FROM
COMBINATORIAL-EXPLOSION   CODING   CODE   COBOL   CLOSED   CLOSE
CHINESE   CHINESE-TOTAL   CHINESE-REMAINDER-THEOREM   CHAIN
CHADLESS-TAPE   CHAD   CEU   CATASTROPHE   CARD   CAMPUS   CALL   CAL
CAI   CAD   BUS   BUNDLES   BUG   BUFFER   BUBBLE-SORT   BUBBLE-MEMORY
BROKET   BREAKPOINT   BOTTOM-UP   BOTTOM-LINE   BOTTOM-DOWN
BOOTSTRAP   BONUS   BLOCKHEAD   BLOCK   BLANK-CARD   BIT-BUCKET   BIT
BINARY-SEARCH   BINARY   BIDIRECTIONAL   BESACK   BENDS   BENCHMARK
BASIC   BASE-ADDRESSS   BALLPARK   BACKUP   BACKTRACKING   BABOL
AUTOEROTICISM   AUGRATIN   ASL   ASCII   ARTIFICAL-INTELLIGENCE   ARPA
ARGUMENT   APPLE   APL   AOS   ANSI   AND   ANCILLARY   ALU   ALLC
ALGORITHM   ALGORISM   ALGORASM   ALGOL-84   ALBOHPHOBIA   ACRONYM
ACK   ABORT   ABM   ABEND   ABACUS    ok
  ok
  ok
  ok
```

Figure 30. Words in the Devil's dictionary (cont'd)

```
FORTRAN
One of the earliest languages of any real height, level-wise,
developed out of Speedcoding by Backus and Ziller for IBM/704
to boost sales of 80-column cards to engineers.


COBOL
COmmon Business Oriented Language.  A procedurally disoriented
language pioneered by Commander Grace Murray Hopper of the U.S.
Navy.  In keeping with the naval tradition, a lot of rum is
still forced down the throats of reluctant middy COBOL
programmers before the swab their daily deck of cards.


ALGOL
Please try the following words:
ALGOL-84  ok
ALGOL-84
An extension of ALGOL being formulated by 84 dissidents from
various ALGOL user groups.



APL
A Personal Language, A Packed Language, or A Programming
Language.  A language, devised by K. Iverson, so compacted that
the source code can be freely disseminated without revealing
the programmer's intentions or jeopardizing proprietary rights.


PASCAL
Please try the following words:

ASCII
ASL
AUGRATIN
AUTOEROTICISM ok
  ok
  ok
  ok
XEROX
Please try the following words:

BABOL
BACKTRACKING
BACKUP
BALLPARK
BASE-ADDRESSS ok
  ok
  ok
BACKUP
Any file, device, or person which results from backing up; the
total deviation from the original is directly proportional to
the number and scale of the catastrophes resulting from each
copying or matching error.  To compound errors while merely
trying to perpetuate them.
```

Figure 31. Sample dialog

```
  ok
  ok
ABACUS
A reliable solid state biquinary computing device now partly
superseded by the Cray series.


IBM
Irish Business Machines Corporation, also called Itty Bitty
Machines, Snow White, The VS Pioneer, The Lawyer's friend.  The
dominant force in computer marketing, having, worldwide,
supplied some 75 percent of all known hardware and 10 percent
of all known software.  To protect itself from the litigious
envy of less successful organizations, such as the U.S.
government, IBM employs 68 percent of all known ex-attorneys
general.


  ok
ANSI
One of supranational bodies devoted to establish standards,
i.e., to change rules which have been universally adopted.


CPU
Central Processing Unit.  The calculating mill that Babbage
dreamed on.


ALU
Arthritic Logic Unit or Arithmetic Logic Unit. A random number
generator supplied as standard with all computer systems.


MEMBRY
Please try the following words:
MAN-HOUR  ok
MAN-HOBR
A sexist, obsolete measure of macho effort, equal to 60
kiplings.  Most areas of DP activity now include a synergistic
mix of male and female operatives, and the man-hour unit is
being replaced by the PERSON-HOUR, using a conversion factor of
1.50.


BASIC
Beginner's All Purpose Symbolic Instruction Code.  Originally,
a simple mid-level lenaguage used to test the students ability
to incremnt line numbers, but now available only in complex,
extended versions.
```

Figure 31. Sample dialog (cont'd)

```
     20
 0 ( DEVIL'S DICTIONARY, CHT, 25-JAN-84)
 1 21 26 THRU   ( Load dictionary utilities.)
 2
 3 DEVIL DEFINITIONS  ( Put dictionary entries in DEVIL.)
 4 60 209 THRU
 5 : WORDS WORDS ;
 6 FORTH DEFINITIONS
 7 27 28 THRU (  Refined name output)
 8 ' SEARCH IS NUMBER   ( Fix the text interpreter.)
 9 PAGE
10 .( Please type a word and hit the return key.)
11 SEAL DEVIL
12 ( Seal other vocabularies.  Leave only DEVIL oper.)
13
14
15
```

```
     21
 0 ( CREATE DICTIONARY, CHT, 14-JAN-84 )
 1 : -CHAR   ( addr n --- addr n1)
 2       ( Remove non-blank trailing characters from a string.)
 3       BEGIN   2DUP 1- + C@ 32 -
 4       WHILE   1-   REPEAT   ;
 5 : ADJUST   ( addr n --- addr1 n1 addr n2 )
 6       ( Extract one line from a string to be printed.)
 7       >R   64 -CHAR   ( --- addr n2 )
 8       2DUP +   ( --- addr n2 addr1 )
 9       OVER R> SWAP -   ( --- addr n2 addr1 n1 )
10       2SWAP   ;
11 : DISPLAY   ( addr count --- )
12       ( Print a long string with word  wrapping.)
13       BEGIN   CR DUP 64 >
14       WHILE   ADJUST -TRAILING TYPE
15       REPEAT   -TRAILING TYPE   ;
```

```
     22
 0 ( CREATE DICTIONARY, CHT, 14-JAN-84 )
 1
 2 : \          ( Dictionary entry compiler-interpreter.)
 3       CREATE  ( Compile a new entry to the FORTH dictionary.)
 4       BLK @ , ( Block #)   >IN @ DUP ,    ( Offset of string)
 5       92 WORD DROP                    ( Scan to the next \ )
 6       -1 >IN +! ( Backup pointer)   >IN @ SWAP - ,  ( Length)
 7       DOES>   ( Runtime interpreter for dictionary entry)
 8       DUP @   DUP SCR !   BLOCK
 9       OVER 2+ @ +   SWAP 4 + @   DISPLAY      ( Print string)
10       CR CR QUIT   ;
11
12 VOCABULARY DEVIL
13
14
15
```

```
     50
( DEVIL'S DICTIONARY, CHT, 25-JAN-84)                    29sep8?
Load dictionary constructor, and soundex searching words.

All DP words are put into this DEVIL vocabulary, which can be
sealed off to protect the underlying Forth system.


This set of words allows the system to display  at few words
when even the soundex search failed.
Replacing NUMBER by the fuzzy SEARCH allows the system to
converse with the user more naturally.  A word  not found in the
dictionary will trigger the fuzzy search and give the user some
useful suggestions.
Seal the DEVIL vocabulary.  Point of no return.
```

```
     51
( CREATE DICTIONARY, CHT, 14-JAN-84)                     29sep8?
-CHAR   ( addr n --- addr n1)
   Remove non-blank trailing characters from a string.
   This word deletes characters of an incomplete word at the end
   of a line.  For word wrap-around.
ADJUST   ( addr n --- addr1 n1 addr n2 )
   Extract one line from a string to be printed.
   addr n2 is the string to be printed in one line.
   addr1 n1 is the remaining string.


DISPLAY   ( addr count --- )
   Print a long string with word wrapping.
```

```
     52
( CREATE DICTIONARY, CHT, 14-JAN-84)                     29sep8?

\  Compile the word following as an entry in the Devil's
   dictionary.  To save space in the memory, the explanation
   is not compiled, but remains in the file.  Only the location
   of the explanation string is stored in the body of the word
   defintion, including the block number and the character
   offset.
   At run time, the explanation string is pulled out of the file
   an printed on the console.


DEVIL   The vocabulary containing all the entries of the Devil's
   dictionary.
```

Listing 16. Devil's dictionary

```
     23
 0 ( FUZZY SEARCH, CHT, 20-JAN-84)
 1
 2 CREATE TABLE   ( Soundex codes for the 26 alphabets )
 3       0 C, 1 C, 2 C, 3 C, 0 C,   1 C, 2 C, 0 C, 0 C, 2 C,
 4       2 C, 4 C, 5 C, 5 C, 0 C,   1 C, 2 C, 6 C, 2 C, 3 C,
 5       0 C, 1 C, 0 C, 2 C, 0 C,   2 C,
 6
 7 : CHAR ( char --- code , convert one character to code)
 8       127 AND DUP 90 > IF 32 - THEN  ( Convert to upper case)
 9       DUP 64 > IF 65 - ELSE DROP 0 THEN
10       TABLE + C@      ;
11
12
13
14
15
```

```
     24
 0 ( FUZZY SEARCH, CHT, 20-JAN-84)
 1
 2 VARIABLE ST  ( last code converted )
 3
 4 : $MOVE   ( dist source --- dist or dist+1 )
 5       C@ 127 AND CHAR   ?DUP IF   Non-zero code)
 6       ST C@ OVER - IF             Non-repeating code)
 7          DUP ST C!   48 - OVER C!   1+
 8       ELSE DROP THEN   THEN   ;
 9
10 : $INIT   PAD 32 46 FILL   0 ST C!   ;
11
12
13
14
15
```

```
     25
 0 ( FUZZY SEARCH, CHT, 25-JAN-84)
 1
 2 CREATE SDX   4 ALLOT
 3
 4 : $CONVERT   ( nfa --- , Convert a string to Soundex in PAD)
 5       $INIT   >R   R@ C@ 31 AND DUP ( Length of string)
 6       R@ 1+ C@ 127 AND   PAD C!   ( Copy the first character)
 7       1 > IF   ( String has more than one character.)
 8       PAD 1+ R>   ROT OVER + 1+ SWAP 2+ DO   I $MOVE   LOOP
 9       ELSE   R> DROP   THEN   DROP   ;
10
11 : $GET   ( addr --- , convert string and move Soundex to SDX.)
12       $CONVERT   PAD SDX 4 CMOVE   ;
13
14
15
```

```
     53
 ( FUZZY SEARCH, CHT, 20-JAN-84)                          29sep84cht
 TABLE    Soundex codes for the 26 alphabets
    0   vowels      a e h i o u w y
    1               b f p v
    2               c g j k q s x z
    3               d t
    4               l
    5               m n
    6               r
 CHAR  ( char -- soundex-code )
    Convert an ASCII character to the soundex code.  Lower case
    are converted to upper case.  Punctuation characters are not
    disturbed.
```

```
     54
 ( FUZZY SEARCH, CHT, 20-JAN-84)                          29sep84cht

 ST  A variable storing the last converted soundex code.
    It is needed in the next conversion as duplicated code must
    be ignored.
 $MOVE   ( dist source --- dist or dist+1 )
    Convert a character at source and add the source code
    to the soundex string at dist.  Increment dist for a valid
    soundex code.  If the code is superfluous, return the dist
    address without change.

 $INIT
    Initialize PAD to receive soundex code.  Reset ST also.
```

```
     55
 ( FUZZY SEARCH, CHT, 25-JAN-84)                          29sep84cht

 SDX   An array holding a soundex string for dictionary search.

 $CONVERT   ( nfa --- , Convert a string to Soundex in PAD.
    Given the name field address of a dictionary entry, convert
    its name to a soundex string and store it in PAD buffer.

 $GET   ( addr --- , convert string and move Soundex to SDX.
    Given a counted string at addr, convert it into soundex
    string and save it into the SDX array.
```

Listing 16. Devil's dictionary (cont'd)

```
     26
 0 ( FUZZY SEARCH, CHT, 25-JAN-84)
 1
 2 : .NAME   ( nfa --- , print name with same Soundex code.)
 3       DUP $CONVERT   PAD 2@ SDX 2@    D- D0=
 4       IF .ID ELSE DROP THEN   ;
 5
 6 : .NAMES   ( lfa --- , list all names with same Soundex codes.)
 7       BEGIN   ?DUP WHILE   DUP 2+ .NAME   @   REPEAT   ;
 8
 9 : SEARCH   ( here --- , fuzzy dictionary search )
10       CR   ." Please try the following words:"
11       CR   DUP CONTEXT @ HASH @   SWAP $EET .NAMES
12       0 ;
13
14
15
```

```
     27
 0 ( PRINT NAMES IN A BLOCK, CHT, 20-APR-84)
 1 FORTH DEFINITIONS
 2 60 CONSTANT MIN-BLOCK  ( Starting block of dictionary text)
 3 208 CONSTANT MAX-BLOCK ( End block of dictionary text)
 4 MIN-BLOCK SCR !
 5 DEVIL DEFINITIONS  ( The new \ must be in the DEVIL vocabulary)
 6 : \   ( Redefine \ to print entry name )
 7       CR   IN @ WORD COUNT TYPE
 8       92 WORD DROP   -1 >IN +!   ;
 9 FORTH DEFINITIONS
10 : NEXT-BLOCK   ( Scan and print names in the next valid block)
11       BEGIN   SCR @ MAX-BLOCK 1- <
12          IF MIN-BLOCK 1- SCR ! THEN
13          1 SCR +!   SCR @ BLOCK C@   92 = UNTIL
14       SCR @ LOAD   ;
15
```

```
     28
 0 ( REFINED SEARCH, CHT, 9-MAY-84)
 1 VARIABLE ?ENTRY   0 ?ENTRY !
 2
 3 : .ID   ( Redefined to increment ?ENTRY.)
 4       1 ?ENTRY !   .ID   ;
 5
 6 26 LOAD ( Recompile .NAME, .NAMES, and SEARCH.)
 7
 8 : SEARCH   ( New search.)
 9       0 ?ENTRY !
10       SEARCH   ( The old SEARCH)
11       ?ENTRY @ 0=   IF NEXT-BLOCK THEN   ;
12
13
14
15
```

```
     56
( FUZZY SEARCH, CHT, 25-JAN-84)                                  29sep84cht

.NAME   ( nfa --- , print name with same Soundex code.)
   Convert the name of a dictionary entry to soundex code.
   If the soundex string matches that stored in SDX, print
   the real name; otherwise, ignore this entry.
.NAMES   ( lfa --- , list all names with same Soundex codes.)
   Go through a linked dictionary thread and print all the names
   which have the soundex code as that in SDX.
SEARCH   ( here --- , fuzzy dictionary search )
   Assuming a counted string is stored on the top of the Forth
   dictionary at HERE, hash the string, find the thread it
   belongs to, search the thread and print all names of the
   soundex code in SDX.
```

```
     57
( PRINT NAMES IN A BLOCK, CHT, 20-APR-84)                        29sep84cht

MIN-BLOCK   Starting block of dictionary text.
MAX-BLOCK   End block of dictionary text.

\   Redefine \ so that it will print the next word instead of
   compiling a new dictionary entry.  This is handy in scanning
   a block of text and print the names of entries defined in
   the block.

NEXT-BLOCK   ( Scan and print names in the next valid block)
   Load the next block pointed to by SCR.  Because \ is redefined
   to print the entry names in a block, loading a block displays
   all the entries defined in this block.
      SCR @ LOAD   ;
```

```
     58
( REFINED SEARCH, CHT, 9-MAY-84)                                 29sep84cht

?ENTRY   True if a dictionary entry is found by the soundex
   search.  False if the word has no match in the dictionary.
   Then, extra help message must be generate.
.ID   ( Redefined to increment ?ENTRY.)
   Increment ?ENTRY if old .ID is called.  Make sure that ?ENTRY
   reflects the state of soundex searching.
26 LOAD ( Recompile .NAME, .NAMES, and SEARCH.)
   SEARCH must also reflect the ?ENTRY status.
SEARCH   ( New search.)
   When the Forth interpreter fails to find a word in the devil
   dictionary, this SEARCH will be invoked in place of NUMBER
   to perform a fuzzy search using soundex code approach.  If
   the soundex searching finds no match, print a few words
   from the file as suggestions to the user to proceed.
```

Listing 16. Devil's dictionary (cont'd)

```
30
( DEVIL'S DICTIONARY, CHT, 25-JAN-84)                      28sep87cht
121 126 THRU   ( Load dictionary utilities.)

\ DEVIL DEFINITIONS  ( Put dictionary entries in DEVIL.)
\ 60 209 THRU

FORTH DEFINITIONS
127 128 THRU  ( Refined name output)
EXIT
' SEARCH 'NUMBER '   ( Fix the text interpreter.)
PAGE
.( Please type a word and hit the return key.)
( Seal other vocabularies.  Leave only DEVIL open.)
SEAL
```

```
0
                                                           29sep87cht
THE DEVIL'S DP DICTIONARY
by Stan Kelly-Bootle
McGraw-Hill Book Company
New York, NY

Adapted to IBM-PC by Dr. C. H. Ting
Based on F83 Forth System. Version 2.1.0.

For the amusement of myself and my friends,
not to be sold or distributed.
```

```
0
                                                           29sep87cht
THE DEVIL'S DP DICTIONARY
by Stan Kelly-Bootle
McGraw-Hill Book Company
New York, NY

Adapted to IBM-PC by Dr. C. H. Ting
Based on F83 Forth System. Version 2.1.0.

For the amusement of myself and my friends,
not to be sold or distributed.
```

```
0
                                                           29sep87cht
THE DEVIL'S DP DICTIONARY
by Stan Kelly-Bootle
McGraw-Hill Book Company
New York, NY

Adapted to IBM-PC by Dr. C. H. Ting
Based on F83 Forth System. Version 2.1.0.

For the amusement of myself and my friends.
not to be sold or distributed.
```

```
0
                                                           29sep87cht
THE DEVIL'S DP DICTIONARY
by Stan Kelly-Bootle
McGraw-Hill Book Company
New York, NY

Adapted to IBM-PC by Dr. C. H. Ting
Based on F83 Forth System, Version 2.1.0.

For the amusement of myself and my friends,
not to be sold or distributed.
```

```
0
                                                           29sep87cht
THE DEVIL'S DP DICTIONARY
by Stan Kelly-Bootle
McGraw-Hill Book Company
New York, NY

Adapted to IBM-PC by Dr. C. H. Ting
Based on F83 Forth System, Version 2.1.0.

For the amusement of myself and my friends,
not to be sold or distributed.
```

Listing 16. Devil's dictionary (cont'd)

```
 60
 0 \ ABACUS A reliable solid state biquinary computing device now p
 1 artly superseded by the Cray series.  \ ABEND A system ABORT de
 2 liberately induced to allow the third shift staff to leave early
 3 .  \ ABM Arab Business Machines.  A shadowy consortium rumored t
 4 o be poised for an IBM takeover bid in the mid-1960's.
 5 \ ABORT  The rather heavy interruption of a job or system, usual
 6 ly self-induced, but sometimes invoked by the user.  To terminat
 7 e in such a manner that future hopes and discussions of revivifi
 8 cation attract predicates in the neighborhood of wishful.
 9 \ ACK A signal indicating that the error-detection circuits have
10  failed.  \ ACRONYM A memorable word from which a non-memorable
11 phrase is acrostically generated.  A circumlocutory abbreviation
12  often confused with its atonym, MNEMONIC.  \ ALBOMPHOBIA The fe
13 ar of palindromes.  \ ALGOL-84 An extension of ALGOL being formu
14 lated by 84 dissidents from various ALGOL user groups.
15

 61
 0 \ ALGORASM A sudden, short-lived moment of pleasure enjoyed by t
 1 he programmer when the final KLUDGE rings the bell.  \ ALGORISM
 2 A pre-LISP ALGORITHM devised by au-Ja'far Mohammed ibn-Musa al-K
 3 huwarizmi who wrote the first BASIC substring modifier in a vain
 4  attempt to shorten hhis name.  \ ALGORITHM A rare species endang
 5 ered by the industry's cavalier pursuit and gauche attempts at d
 6 omestication.  \ ALLC Association for Literacy and Linguistic Co
 7 mputing.  An international association founded by Prof. Rov Wisb
 8 y and Mrs. Joan Smith to promote the use of SNOBOL.  \ ALU Arthm
 9 tic Logic Unit or Arithmetic Logic Unit. A random number generat
10 or supplied as standard with all computer systems.  \ ANCILLARY
11 ancilliary, essentail.  The primary purchase is designed to gene
12 rate a growing list of essentail adjuncts.  \ AND To conjunct sev
13 eral binary victims in the boolean environment.  \ ANSI One of su
14 pranational bodies devoted to establish standards, i.e., to chan
15 ge rules which have been universally adopted.

 62
 0 \ AOS To increase the amount of something, as:"They aossed my ho
 1 urs but sossed may pay."  \ APL A Personal Language, A Packed La
 2 nguage, or A Programming Language.  A language, devised by K. Iv
 3 erson, so compacted that the source code can be freely dissemina
 4 ted without revealing the programmer s intentions or jeopardizin
 5 g proprietary rights.  \ APPLE A popular personal computer with a
 6  refreshing nonnumeric, nonacronymic apple-ation.  \ ARGUMENT AA
 7 disputatious variable constantly pouncing on innocent FUNCTIONS.
 8  \ ARPA Advanced Research Projects Agency.  An agency of the U.S
 9 . Departement of Defense estaablished in 1968 to test its defens
10 es against misuse and piracy in the large-scale distributed proc
11 essing environment.  \ ARTIFICAL-INTELLIGENCE The misguided searc
12 h for a lower unit-cost Homo sapiens at a time when the majority
13  of the species remains critically underexploited.  The construc
14 tion of algorithms for the blackleg assembly of wooden building
15 block motor cars.

 63
 \ ASCII  American Standard Code for Information Interchange.  A 7
 - or 8-bit code forced upon the free world by vicious anti-IBM r
 ebels, led by the U. S. Government, who held 16 card-carrying SD
 CDIC hostages at gunpoint in a Washington committee compound for
  two years.  \ ASL American Sign Language A formal system of bod
 signs for use in nonverbal, interpersonal communications envir
 nment.  \ AUGRATIN Amalgamated Union of General Rewriters, Amende
 rs, Tinkerers, and INterpolators.  See PAYROLL.  \ AUTOEROTICISM
 The computer generation of best-selling novels.

 64
 EXIT

 65
 EXIT
```

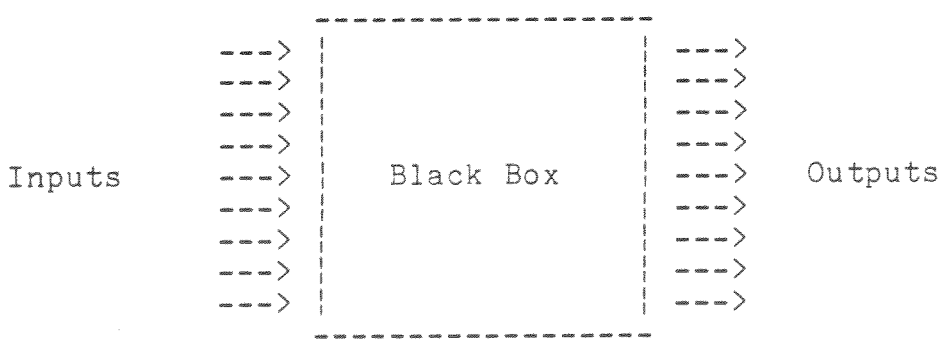
Listing 16. Devil's dictionary (cont'd)

XVII. NEURAL NETWORKS AND EXPERT SYSTEMS

1. KNOWLEDGE REPRESENTED AS MATRICES

Matrices are shorthand notations best suited to represent multivariable algebraic equations and linear transformations. They allow complicated physical and engineering problems to be described concisely and solved conveniently. Here I shall discuss the use of matrices to represent knowledge as needed for many different fields in the artificial intelligence.

At a very high and concepture level, a computer or a large class of computing devices can be viewed as a black box which accepts a set of inputs and produces a set of outputs:



The class of problems best solved by this black box is the 'time independent' problems, where a set of inputs always produces a definite set of outputs, independent of the past history of the system. This is similar to a group of logic gates without memory.

This black box can be represented by a matrix. The operation to transform the inputs into appropriate outputs is a matrix multiplication. Let the inputs be a vector {x1, x2, ..., xm}, the black box be a matrix {a11, a12, ..., a1m, a21, a22, ..., a2m, ..., an1, an2, ..., anm}, and the outputs be another vector {b1, b2, ..., bn}, the multiplication is then:

$$Ax = b$$

This is the Hopfield model of a neural network, if we assumes that an input value has two states, either on or off, and that the an output value is further evaluated through a threshold circuit to produce an on or off state, corresponding to the firing or inactive state of a neuron.

In this neural network, the knowledge of the system is contained in the matrix A, which completely determines the behavior of the network. The matrix may be determined by training or by prior information about the problem. No matter how the matrix is derived, the network will behave the same once the matrix is fixed.

The black box approach is also applicable to many problems which use expert systems to solve. Most expert systems are based on rules, which are the basic units of knowledge. A rule generally is presented in the form of an IF-ELSE-THEN structure. However, a subject matter usually contains many such rules. If the rules are time independent, a collection of related rules can be reformatted to become a gigantic multiple input case structure. Some examples are:

IF (feature, beak, can-fly, lay-eggs) THEN bird.
IF (carriage, can-fly, engine) THEN airplane.

The inputs can be represented by a column vector with logic values as elements, and the output will be one element in another column vector which indicates the outcome of the logic selection by the input vector. The selection rules can be condensed into a matrix and the selection process becomes a multiplication of this matrix by the input vector. The largest element in the resulting vector product gives the outcome.

It is probably very wasteful of memory to represent a large expert system with a single matrix. If the expert system can be divided into subsystems and submodules, it can be more efficiently represented by a number of small matrices organized in a tree structure. The outcome of a high level matrix selects one sub-matrix at a lower level. The selection proceeds until the lowest level matrix yields a useful conclusion.

2. NEURAL NETWORK SIMULATOR

A neural network simulator is shown in Listing 17. Screen 6 shows the words defining the memory arrays. This network can accommodate 256 inputs and 256 outputs. There are thus up to 64K synapes in the network. The inputs are stored in a byte array INPUT, which is 256 bytes long, and the outputs are stored in a number array SUMS, which has 512 bytes. The synapses are stored in a 64 Kbyte data segment, 20000H to 2FFFFH in the IBM-PC. The weight factor of a synapse is represented by a byte, with a range of -128 to 127. The input data are either binary, with two values of 1 and 0, or bipolar, with two values 1 and -1. The output of a neuron is the sum of the products between inputs and the corresponding weights in the synapses. This sum can be represented by a 16 bit two's complement number, without possibility of overflow.

Within this huge network, a practical application is defined by the number of inputs and the number of neurons. ROWS is

the variable containing the number of neurons and COLUMNS is the variables containing the number of inputs. The number of inputs includes feedbacks from neurons. INPUT# is the number of inputs from the environment. Generally, COLUMNS equals to the sum of INPUT# and ROWS.

In cases where a threshold value should be subtracted from the sum to determine whether a neuron will fire or not, a 256 number array THRESHOLD is also defined to store a 16 bit threshold value for each neuron. The word DISPLAY prints the weights in the network on the CRT screen.

The neural network can thus be shown schematically as in Figure 32.

PRODUCT in Screen 7 evaluates the products at the synapses of a neuron and sums them to SUMS. The input is assumed to be binary. If the input term is not 0, the weight at the synapse is summed; otherwise, the weight is ignored.

FEEDBACK takes the values in SUMS and deposite 0 or -1 into the feedback part of the array IMPUT. These values will be used in the next round of neuron processing.

The words in Screen 8 are services to set up and to display the network. .SUMS and .INPUT print the corresponding arrays. ROW! takes a row of weights and stores them into a row in the network. COLUMN! does similar thing for a column.

Screen 9 shows the neural network of a 4 bit A/D converter, as described by Hopfield and Tank(1). Here column 0 is the threshold, column 1 is the input voltage, and columns 2 to 5 are the weights in the neural network. To exercise this network, one first gives it an input voltage by the commands

```
RESET  5  VOLT
```

which set appropriate values in column 1 and initialize the input array. Then, XX will take the input and evaluate the outputs, which is then feedback to the input array. XX can be executed repeatedly to get the network to converge.

Using integer arithmetics, most time the output swings between 0 and 15, instead of settling at the proper binary pattern corresponding to the input voltage. However, if the correct pattern is set up in the input array, the network locks to that pattern correctly.

Another example is shown in Figure 17. It is taken from a study of bacteria identification(2). 16 standard tests are performed on a culture of bacterium. The color after reaction in each test can be recorded as positive or negative to this test. The results of the 16 tests as a 16 bit pattern can then be used to match against the entries in a table of 54 different bacteria. The closest matches will then be reported to the operator as possible identifications.

For some bacteria, the result of a test may be positive, negative, or not significant. These three cases are represented by 1, -1, or 0 as the weight in the table. The test input can also take these three value, meaning that the test is positive, negative, or ambiguous. Thus the product rule has to be modified as shown in Screen 12. The weight table is filled by loading Screens 13 to 16.

A test pattern is loaded into the input array by !INPUT, and the matching results can be computed and displayed by the word XX. The largest sum displayed corresponds to the closest bacteria identification.

The same technique can be used for many other applications. One interesting example is identifying characters printed by dot matrix printer. Most dot matrix printer prints characters in a 8 by 8 dot matrix, which can be mapped into an array of 8 bytes, or 64 bits. To differentiate 128 characters encoded in the 8 by 8 bit matrix, we can use a neural network of 64 inputs and 128 outputs. If an input pattern matches perfectly with a character known to this network, we will get 64 matched bits. Most characters differ by more than 10 dots; therefore, there are adequate margins to recongize characters in noisy environment in which quite a number of dots can be missing or misplaced.

There are much more efficient ways to compare 64 bit patterns. The program in Screen 41 of Listing 17 is my version. The word MATCHING takes two 8 byte arrays at addr1 and addr2 and counts the number of matching bits in these two arrays. The matched bit count is stored in the variable BITS. Data in these arrays are fetched in 16 bit chunks and are XOR'ed. The bits set by XOR operation is then used by COUNT-BITS to increment the bit count in BITS. The dot matrix patterns of the 128 characters are stored in Block 98, and the 8 byte pattern to be matched is stored in PAD buffer. The number of un-matched bits against one of the 128 stored patterns is stored back in the 128 bytes RANK array. The smallest entry in RANK corresponds to the character best matched to the pattern in PAD. SELECT scans through the RANK array and returns the position of the smallest entry.

3. AN EXAMPLE OF EXPERT SYSTEM

A rule based expert system is composed of an inference engine and a knowledge base. The knowledge of the expert is generally encoded as a set of rules, which are a collection of some IF-ELSE-THEN structures. The inference engine tries to traverse this knowledge base, with user input to guide the traversal. At the end, the expert system will either reach a point where a definite consequence can be concluded or no advice can be offered because the particular knowledge was not encoded into the knowledge base.

Traversing through the knowledge base generally is very time consuming, especially when the knowledge base is stored in text form. Compiling the knowledge base into some standard record structure would let the inference engine run much faster.

From a macroscopic point of view, an the knowledge base is a gigantic case structure, with multiple inputs and multiple outcomes. A specific set of inputs will produce a specific outcome.

In a time independent system, where the outcome depends only on the instentaneous state of the inputs, the case structure can be very conveniently expressed in the form of a matrix. This matrix has as many columns as there are inputs, and as many rows as the outcomes. The inference engine thus takes a set of inputs and matches this input pattern with the patterns in the matrix, stored in rows. The best match determines the outcome as the proper answer. The matching process could be summing the matching bits in a row, or a multiplication of the input as a column vector with the matrix. The largest element in the product column selects the outcome.

A rather trivial example is show in Figure 33 below. From a list of features, the expert system can determine whether the object under consideration is a dog, cat, bird or an airplane.

Figure 33. A Trivial Expert System

Feature	Wing	Carr-iage	Fea-ther	Beak	Eng-ine	Hair	Scale	Craw	Swin	Fly
Airplane	1	1	-1	-1	1	-1	-1	-1	-1	1
Glider	1	1	-1	-1	-1	-1	-1	-1	-1	1
Bird	1	-1	1	1	-1	-1	-1	-1	-1	1
Fish	-1	-1	-1	-1	-1	-1	1	-1	1	-1
Cat	-1	-1	-1	-1	-1	1	-1	1	-1	-1
Dog	-1	-1	-1	-1	-1	1	-1	1	1	-1

If we use the row of Airplane as input vector, and multiply it with the matrix in Figure 4, the resulting vector is {10, 8, 2, -2, -2, -4), showing that the best match is the airplane as expected. It also shows that Glider matches quite well with airplane while other animals are very different from an airplane. Examining the product values in the results vector allows one to evaluate a range of alternative selections and thus more useful than a straight selection of a single outcome.

This approach also allows that the matrix element be expressed in fractions between -1 and +1 to accomodate fuzzy logic operations. This provision is important in cases where features cannot be determined with certainty but have to be expressed in terms of probability and statistics.

```
6printing off 6PRINTING

    Page# 1        B:NEURAL.BLK


      6
 0 \ matrix sum
 1 create input 256 allot   input 256 erase
 2 create sums  512 allot   sums 512 erase
 3 create threshold 512 allot   threshold 512 erase
 4 variable input#   5 input# !
 5 variable rows   5 rows !
 6 variable columns   10 columns !
 7 hex 2000 constant dseg decimal
 8 : display
 9    cr 4 spaces   columns @ 0 do  i 4 .r loop
10    rows @ 0 do   cr  i 4 .r
11       columns @ 0 do   i j flip or
12       dseg (x@) dup 128 and if -128 or then  4 .r
13    loop   loop   ;
14
15


      7
 0 \ product, feedback
 1 : product
 2    rows @ 0 do   0   columns @ 0 do
 3       j flip i + dseg (x@)   dup 128 and if -128 or then
 4       i input + c@      0> if + else drop then
 5       loop   i 2* dup >r   threshold + @ -   r> sums + !
 6    loop ;
 7 : feedback
 8    rows @ 0 do
 9       i 2* sums + @   0>   i input + input# @ + c!
10    loop ;
11
12
13
14
15


      8
 0 \ show results
 1 : .sums   rows @ 0 do   i 2* sums + @ 3 .r   loop ;
 2 : .input   columns @ 0 do   i input + c@ 3 .r loop ;
 3 : row! ( ... row -- )
 4    flip   columns @ 0 do
 5       swap over i + dseg (x!)
 6    loop drop ;
 7 : colum! ( ... colum -- )
 8    rows @ 0 do
 9       swap over i flip + dseg (x!)
10    loop drop ;
11 : volt ( n -- )
12    >r   r@ 8 * r@ 4 * r@ 2*   r>   1 colum! ;
13 : reset   257 input !   input 2+ 4 erase ;
14 : xx      product .sums feedback ;
15


      9
07may87cht \ a/d converter                                    07may87cht
           4 rows !
           6 columns !
           2 input# !
           -8 -4 -2 0 1 0      0 row!
           -16 -8 0 -2 2 -2    1 row!
           -32 0 -8 -4 4 -8    2 row!
           0 -32 -16 -8 8 -32  3 row!
           exit
           -16 -8 -4 0 2 -1       0 row!
           -32 -16 0 -4 4 -4      1 row!
           -64 0 -16 -8 8 -16     2 row!
           0 -64 -32 -16 16 -64   3 row!


      10
06may87cht \s                                                 08may87cht


      12
08may87cht \ bacteria identification                          08may87cht
           : product
              rows @ 0 do   0   columns @ 0 do
                 j flip i + dseg (x@)   dup 128 and if -128 or then
                 i input + c@
                 dup if   1- if - else + then   else drop then
                 loop   i 2* dup >r   threshold + @ -   r> sums + !
              loop ;
           : xx   product .sums ;
           : !input    columns @ 0 do   input i + c! loop ;
           \ : row!   cr 3 .r 2 spaces !input xx ;
           54 rows !   18 columns !
```

Listing 17. Neural network

```
 13
 0 \ bacteria data.
 1 -1 -1  1 -1 -1 -1  1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 2 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1  0  0 -1 -1 -1 -1
 3 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1
 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1
 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 6 -1  1  1 -1  1 -1  1 -1 -1 -1  1 -1 -1  1 -1 -1 -1 -1
 7 -1 -1  1  1  1 -1  1 -1 -1  1  1 -1  1  0 -1 -1 -1 -1
 8 -1 -1  1  1  1 -1  1 -1 -1  1  1 -1  1  1 -1 -1 -1 -1
 9 -1 -1  0 -1 -1 -1  1 -1 -1 -1  1 -1 -1  1 -1 -1 -1 -1
10 -1 -1 -1 -1 -1 -1  1 -1 -1 -1  1 -1  1  1 -1 -1 -1 -1
11 -1  1  1 -1 -1 -1  1 -1 -1 -1  1 -1  0  1 -1 -1 -1 -1
12 -1 -1  1 -1 -1 -1  1 -1  0  0  1 -1  1  1 -1 -1 -1 -1
13 -1  0  1 -1 -1 -1  1 -1 -1  1  1 -1  1 -1 -1 -1 -1 -1
14 -1 -1 -1 -1 -1 -1  1 -1 -1 -1  1 -1  1  1 -1 -1 -1 -1
15 -1 -1  1  1  1 -1  1 -1 -1  1  1 -1  1  1 -1 -1 -1 -1
```

```
 14
 0 \ bacteria data
 1 -1 -1  0  1  1 -1  1 -1 -1 -1  0 -1  1 -1 -1 -1 -1 -1
 2 -1  0  1  1  1 -1  1 -1 -1 -1  1 -1  1  1 -1 -1 -1 -1
 3  1 -1  1  1  1 -1 -1 -1 -1 -1  1 -1  0  1 -1 -1 -1 -1
 4 -1 -1 -1  1  1 -1 -1 -1 -1 -1  1 -1  0  1 -1 -1 -1 -1
 5 -1 -1 -1  1  1 -1  1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1
 6 -1 -1 -1 -1 -1 -1  1 -1 -1  1 -1 -1  1 -1 -1 -1 -1 -1
 7 -1 -1 -1 -1  0 -1  1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1
 8 -1 -1 -1  1  1 -1  1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1
 9 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1
10 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1  1  1 -1 -1 -1 -1 -1
11 -1 -1  1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
12 -1 -1  1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
13 -1 -1  0 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
14 -1 -1  0 -1 -1 -1  1 -1 -1  0 -1 -1  1 -1 -1 -1 -1 -1
15 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
 15
 0 \ Extend a Chain
 1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1
 2 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 3 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 4 -1 -1 -1 -1 -1  1 -1  1 -1 -1 -1 -1 -1 -1 -1  0 -1 -1 -1
 5 -1 -1 -1 -1  1 -1  1 -1 -1 -1 -1 -1  1 -1 -1  1 -1 -1 -1
 6 -1  1  1  1  1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 7  1  0  0  1  1  1  1 -1 -1 -1  1 -1 -1  1 -1 -1 -1 -1 -1
 8  1 -1 -1  1  1 -1  1 -1 -1 -1  1 -1  0  0 -1 -1 -1 -1 -1
 9  1 -1 -1  1  1 -1  1 -1 -1 -1  1 -1  0  1 -1 -1 -1 -1 -1
10  1  1  1  1  1 -1  1 -1 -1 -1  1 -1  0  1 -1 -1 -1 -1 -1
11  1 -1  0  1  1 -1  1 -1 -1 -1  1 -1  0  1 -1 -1 -1 -1 -1
12 -1 -1  0  1  1 -1  1  0 -1 -1  1 -1  1  1 -1 -1 -1 -1 -1
13 -1  1  0  1  1 -1  1 -1 -1  0  1 -1 -1  0 -1 -1 -1 -1 -1
14  1  1  1  1  1 -1  1 -1 -1  0  1 -1  0  0 -1 -1 -1 -1 -1
15 -1 -1  0  1  1 -1 -1 -1 -1 -1  1 -1 -1  1 -1 -1 -1 -1 -1
```

```
 16
08aay87cht \ Extend a Chain
 0 row!  -1 -1  1  0  1 -1  1  1 -1 -1  1 -1 -1  0  1 -1 -1 -1
 1 row!  -1 -1  0 -1 -1 -1  1 -1 -1 -1  1 -1 -1  0 -1 -1 -1 -1
 2 row!  -1 -1  1  1 -1 -1  1 -1 -1 -1  1 -1 -1  0 -1 -1 -1 -1
 3 row!  -1 -1 -1 -1 -1 -1  1 -1 -1 -1  1 -1 -1  0 -1 -1 -1 -1
 4 row!  -1 -1 -1 -1  0 -1  1 -1 -1 -1  1 -1 -1  0 -1 -1 -1 -1
 5 row!  -1 -1  1  1 -1 -1  1 -1 -1 -1  1 -1 -1  0 -1 -1 -1 -1
 6 row!  -1 -1  1 -1 -1 -1  1 -1 -1 -1  1 -1 -1  0 -1 -1 -1 -1
 7 row!  -1  1  1  1  1 -1  1 -1 -1  0  1 -1  1  1 -1 -1 -1 -1
 8 row!  -1 -1 -1  1  1 -1  1 -1 -1 -1  1 -1 -1  0 -1 -1 -1 -1
 9 row!
10 row!
11 row!
12 row!
13 row!
14 row!
```

```
08aay87cht
45 row!
46 row!
47 row!
48 row!
49 row!
50 row!
51 row!
52 row!
53 row!
```

0

0

Listing 17. Neural network (cont'd)

```
     40
 0 \ Display characters                                  05jul87cht
 1 : row ( addr c -- )
 2    at   c@   8 0 do dup dup 128 and if 7 else 32 then
 3       emit   2* loop   drop ;
 4 : brows     ( addr -- )
 5    8 0 do   dup 3 .r   2over 1 +      . addr addr'   ."
 6       row   loop   2drop drop ;
 7 : .char ( n -- char -- )
 8    8 * 1023 and 96 block +   brows ;
 9 : first ( char -- )   0 3 rot .char ;
10 : second ( char -- )   16 3 rot .char ;
11 : third ( char -- )   32 3 rot .char ;
12 : fourth ( char -- )   48 3 rot .char ;
13 : fifth ( char -- )   64 3 rot .char ;
14 : window   0 15 at-line   0 23 at #out off ;
15 : display   dark   ." window is cr ;
```

```
     41
 0 \ character matching                                  05jul87cht
 1 variable bits
 2 create rank 128 allot
 3 variable best
 4 : count-bits ( n -- , accumulate set bits into 'bits' )
 5    16 0 do   dup 1 and if 1 bits +! then   2/ loop drop ;
 6 : matching ( addr1 addr2 -- , accumulate into bits )
 7    bits off   4 0 do   over @ over @ xor count-bits
 8       2+ swap 2+ swap   loop 2drop ;
 9 : ranking ( -- )
10    128 0 do   i 96 * 96 block + pad matching
11       bits @   rank i + c!   loop ;
12 : select ( -- n, best matched character )
13    best off   0   128 0 do   pad i + c@ dup   best @ >
14       if best !   drop i   else drop then loop ;
15
```

```
     42
 0 \ print match table                                   10jul87cht
 1 : print-rank ( rank-addr -- )
 2    40 0 do   dup i + c@ 3 .r   ['] ev! abort" *"   loop drop ;
 3 : first-panel ( addr -- )
 4    128 0 do   dup pad 8 cmove   ranking
 5       cr   i 2 .r 3 spaces   rank print-rank
 6    8 +   loop   drop ;
 7 : second-panel ( addr -- )
 8    128 0 do   dup pad 8 cmove   ranking
 9       cr   i 2 .r 3 spaces   rank 40 + print-rank
10    8 +   loop   drop ;
11 : third-panel ( addr -- )
12    128 0 do   dup pad 8 cmove   ranking
13       cr   i 2 .r 3 spaces   rank 80 + print-rank
14    8 +   loop   drop ;
15
```

```
     43
 0 \ transposing apple characters                        10jul87cht
 1 : byte-transpose ( b1 -- b2 )
 2    0 swap   8 0 do   dup 8 and if swap 4 + swap then
 3       2/ swap 2* swap loop   drop ;
 4 : char-transpose ( addr1 addr2 -- , transpose pattern1 to b2 )
 5    dup 8 erase   swap if* swap
 6    7 0 do   over i + c@   byte-transpose   over i + c!
 7       loop 2drop ;
 8 : block-transpose ( src-blk dst-blk -- )
 9    block swap block swap
10    1024 0 do   over i +   over i +   char-transpose
11       8 +loop   2drop ;
```

```
     44
 0 \ Set character set table                             05jul87cht
 1 \ 43 47 thru   extended addressing)
 2 code cseg ( -- cseg)
 3    cs ax mov   ax push   next end-code
 4 : ,block ( d block -- )
 5    block 0   cseg @ d2* d2* d2* d2* c+
 6    1024 move update ;
 7 \ hex ffaba, b2 ,block ( copy 128 characters to block 98)
 8 \ ffaba, b3 ,block   copy next 128 characters to block 99)
```

```
     45
 0 \ Move string between segments                        08Sep87cht
 1 code (XMOVE)   ( s1 s0 d1 d0 # --- , intersegment cmove )
 2    CLD   SP AX MOV   ES BX MOV   DS DX MOV
 3    CX POP   ES POP   DI POP   DS POP   SI POP
 4    REP BYTE MOVE
 5    AX SP MOV   BX ES MOV   DX DS MOV
 6    NEXT END-CODE
 7 : XMOVE   ( source-d dest-d # --- , absolute cmove )
 8    >R . 16 */MOD DROP   2SWAP 16 */MOD DROP
 9    2SWAP R>   (XMOVE)   ;
10 BYE
11 Absolute addressing is using 32 bit addresses instead of
12 8086 specific segment addresses.
```

Listing 17. Neural network (cont'd)

```
     46
 0 \ Fetch and Store characters                                  0
 1 CODE (X@)   ( si ds --- c , intersegment C@)          05Sep85cht
 2    IP BX MOV   DS DX MOV   DS POP   SI POP
 3    AX AX SUB   AL LODS
 4    BX IP MOV   DX DS MOV   IPUSH   END-CODE
 5 : X@   ( d --- c , absolute C@ )
 6    16 MU/MOD DROP   (X@)   ;
 7 CODE (X!)   ( c di es --- , intersegment C! )
 8    ES DX MOV   ES POP   DI POP   AX POP
 9    AL STOS   DX ES MOV
10    NEXT   END-CODE
11 : X!   ( c d --- , absolute C!  )
12    16 MU/MOD IROP   (X!)   ;
13
14
15


     47
 0 \ XFILL and XDUMP                                             0
 1 CODE (XFILL) ( di es # c --- , intersegment FILL )    05Sep85cht
 2    ES DX MOV   AX POP   CX POP   ES POP   DI POP
 3    REP AL STOS   DX ES MOV   NEXT   END-CODE
 4 : XFILL ( d # c --- , absolute FILL )
 5    ISWAP 16 MU/MOD DROP ISWAP   (XFILL)   ;
 6 : XDUMP  ( d # --- , absolute DUMP )
 7    0 DO   DS IDUP 5 U.R   2 SPACES
 8       16 0 DO   IDUP C@ 3 .R   1+ LOOP
 9    16 +LOOP   IDROP   ;
10
11
12
13
14
15


     0                                                           0
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Listing 17. Neural network (cont'd)

FORGET

XVIII. CHINESE LIMERICKS

After reading Nathanial Grossman's paper '7776 Limericks' which appeared in Forth Dimensions Volume 8, No. 6, p. 28, March/April 1987, my immediate reaction was that Chinese is the best language for computerized limericks. The reasons are:

1. Chinese words are single characters, which can be arranged neatly to form sentences of fixed length to construct poems.

2. All words are single syllable in pronunciation. Metric and rhyme can be easily arranged.

3. A vast literature base is available. The 5 character and 7 character poems of Tang Dynasty are especially suited for computerization.

Although I was fully convinced that much more serious and interesting limericks can be derived from Tang poetry, I was not able to do anything then, because I did not have enough tools to implement them. I had a Chinese character generator board which turned an Apple II computer to a elementary Chinese typewriter. I use it to compose and print bulletins and announcement for our church. The program has to be written using Applesoft BASIC, and words and sentences could not be manipulated easily. Another problem is that each Chinese character is encoded to one to five ASCII characters and terminated by a blank character. This is the Chang-je Chinese character coding method, which was one of the best system in encoding and constructing Chinese characters, very popular in Taiwan and Southeast Asia. The variable length representation of Chinese characters make it very difficult to store and manage large amount of text.

The possibility of implementing Chinese limericks was a very important factor motivating me to upgrade my IBM clone system. My clone was one of the first systems built by my pirate friends in Taiwan. It has 256K of memory, 2 floppy drives, and a color card. It has served me well for more than 3 years. Using Forth almost exclusively, I have never encounter a situation in which I need more memory or hard disk.

This summer, I found a very good Chinese word processing system for IBM PC/XT/AT, by Kuo Chiao Business Company in Taipei. It contains about 6000 most commonly used Chinese characters in two fonts, 16 by 16 dots and 24 by 24 dots. The 24 by 24 font has near typeset quality. However, it requires 640K of memory and a hard disk to store and use the fonts. After much soul searching and consultation with members in our church, we reached the conclusion that it is time to catch up with the technology and upgrade the PC so that we can use the Chinese system, code name

KC500.

KC500 works very smoothly. It comes with a WordStar-like word processor so that we do not have to write programs to compose and print bulletins. It also allows the user four options of input methods to select Chinese characters: the Chang-je method which is based on radicals and strokes, the Standard Pronunciation method based on the phonetic spelling, the Inner Code method based on the 2 byte coding system developed in Taiwan, and the Simplified Chang-je method, which uses the first and the last two codes in Chang-je method and a menu to select a Chinese character.

The best feature of KC500 is the way it stores Chinese text files. The Chinese characters are stored in Inner Code, two bytes per character. This feature makes programming limericks very easy, because fixed length verses can be stored and retrieved very conveniently to/from the text files generated by the word processor.

From all the different styles of Chinese poems, I chose the so called 'five character old poem' style of early Tang Dynasty. In this style, each verse consists of 5 characters and each poem contains 4 to 2n verses, where n can be over one hundred. Rules on tonal inflection, rhyme, and symmetry of words in verse pairs are not held as strictly as in the 'ruled poems' in the middle and late Tang period. I also restrict myself to the work of Li Po, the most famous poet in Tang Dynasty, because I happened to own a copy of his selected poems.

I keyed in 1082 verses of the 5 character poems in this book by Li Po, forming the data base of the limericks. A few samples are shown in Figure 34. Here poems are separated by two carriage return and line feed pairs. In the data base, only one carriage return/line feed pair is retained, as in between verses. Each record in the data base is 12 bytes long, ten bytes for the five characters and a carriage return and a line feed. The CR/LF pair is convenient in listing the verses and printing the data base.

The limerick program is the Forth word POEM defined in Screen 2 of Listing 18. It picks four verses randomly and prints them on the CRT screen. Given a verse number, the word .VERSE locates that verse in the data base and prints that verse, followed by a carriage return and line feed. KC500 captures the ASCII character sequence before they are emitted to the screen. If a pair of ASCII characters are a valid inner code of a Chinese character, this character is printed using the font stored in memory; otherwise, the individual ASCII characters are emitted.

The word CHAR takes a byte offset as input, fetches that byte from the data base file, and then emits it. Each byte must be fetched individually from the file and emitted. It might be more convenient to TYPE 12 bytes at once, but one gets into trouble when a verse crosses the block boundary.

A few samples of limerick are shown in Figure 35, produced by POEM.

LIMERICKS in Screen 3 is a flashy demonstration of the limericks. It first clears the CRT display, opens a rectangular window randomly on the screen, prints a poem in the window, and then continues. A long delay is thrown in so that people have time to read the poem (if he can read Chinese.) The results after a number of poems are displayed are shown in Figure 36.

WINDOW.VERSE prints one verse in the window with two leading blanks and two trailing blanks. CLEAR displays 9 white squares forming one row of background in the window. BOX displays a 9x6 window of white square, ready for the poem to be revealed at the center of the window. WINDOW picks a random location on the CRT screen, opens a window there and displays a poem.

EVERYTHING dumps the entire data base to the screen, with four verses to a line. This is used to print the data base on the printer.

This limerick program at this stage is really dumb. It only knows how to display a poem. I am in the progress to classify the verses according to the rhyme of the last character. If verses are chosen from the same class, they will rhyme properly. If the verses are further classified according to their contents, limericks can then convey some meaningful thoughts. As we divide the data base into finer partitions, we will need more verses in each partition. More poems from other poets can then be added to the data base.

A more ambitious undertaking will be generating limericks in the 'ruled poem' style. There are eight verses in a ruled poem. Verses 3 and 4, and verses 5 and 6 are two pairs of symmetric or mirrowed verses. Words in a symmetric pair must have the same grammatical structure. The meaning of words or phrases in the contrasting verses must be similar or opposite. Using these rules, verses can be reconstructed with classified words and phrases. This will greatly increased the possible combinations to construct limericks.

Chinese is the last language to be computerized, because of the sheer mass of characters. Once it is conquered, there are lots of interesting thing we can do with it. Poetry is one area where computered can help in organizing the data and making the information readily available. This limerick program only scratches the surface of the possibilities.

縱看成嶺側成峰
遠近高低各不同
不識廬山真面目
只緣身在此山中

春花秋月何時了
往事知多少
小樓昨夜又東風
故國不堪回首月明中
雕欄玉砌今猶在
只是朱顏改
問君能有幾多愁
恰似一江春水向東流

朝辭白帝彩雲間
千里江陵一日還
兩岸猿聲啼不住
輕舟已過萬重山

月落烏啼霜滿天
江楓漁火對愁眠
姑蘇城外寒山寺
夜半鐘聲到客船

西登香爐峰
南見瀑布水
挂流三百丈
噴壑數十里
簸如飛電來
隱若白虹起
初驚河漢落
半灑雲天裡
仰觀勢轉雄
壯哉造化工
海風吹不斷
江月照還空
空中亂灑射
左右洗青壁
飛珠散輕霞
流沫沸穹石
而我遊名山
對之心益閑
無論激瓊液
且得洗塵顏
且諧宿所好
永遠辭人間

昨日登高罷
今朝更舉觴
菊花何太苦
遭此兩重陽

亭伯去安在
李陵降未歸
愁容變海色
短服改胡衣

談笑三軍卻
交遊七貴疏
仍留一雙箭
未射魯連書

早起見日出
暮看樓烏還
客心自酸楚
況對木瓜山

送客謝亭北
逢迴縱酒還
屈盤戲白馬
大笑上青山
迴鞭指長安
西日落秦關
帝鄉三千里
杳在碧雲間

江城如畫裡
山晚望晴空
兩水夾明鏡
雙橋落采虹
人煙寒橘柚
秋色老梧桐
誰念北樓上
臨風懷謝公

待月月未出
望江江自流
候忽城西郭
青天懸玉鉤
素華雖可攬
清景不同遊
耿耿金波裡
空瞻支鵲樓

Figure 34. Examples of Chinese poems

C> ok
poem

眉目豔量月
覺來汚庭前
挂流三百丈
東風吹愁來
ok
poem

李陵沒胡沙
曲盡己忘情
天地即裴枕
眾鳥高飛盡
ok
poem

屐上足如霜
玉階生白露
仰觀勢轉雄
屈盤戲白馬
ok
poem

髻如雲中鳥
杳在碧雲間
青青蘆葉齊
何異太常妻
ok
poem

人聞桂花落
仍留一隻箭
夜臺無曉日
疑是地上霜
ok
poem

思歸但長嗟
三百六十日
皓齒信難開
髻如雲中鳥
ok

poem

素華雖可攬
如聽萬壑松
南見瀑布水
西施越溪女
ok
poem

愁殺蘋舟人
時鳴春澗中
玉壺繫青絲
只在此山中
ok
poem

兩水夾明鏡
勞勞送客亭
愁見雪如花
春草如有意
ok
poem

餘響入霜鐘
長干吳兒女
正好銜盃時
長嘯絕人群
ok
poem

欲窮千里目
夜臺無曉日
對之心益閑
涉江弄秋水
ok

Figure 35. Chinese limericks generated by POEM

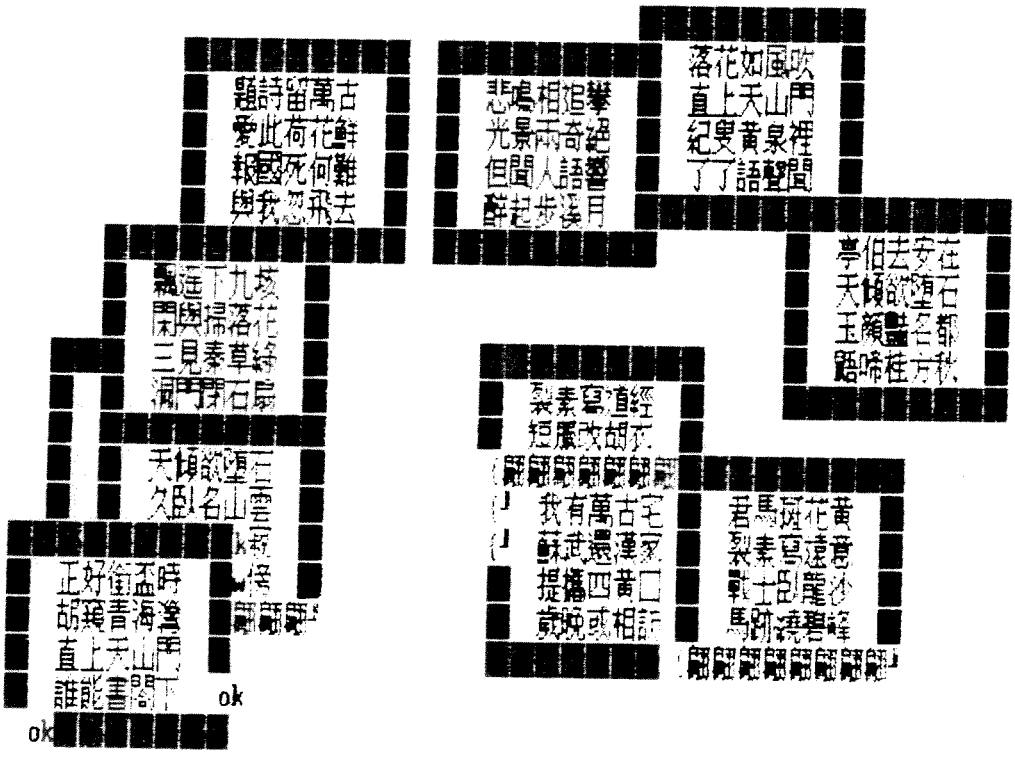


Figure 36. Screen display of limericks

```
      1
 0 \ Chinese limericks                                  16aug87cht
 1 2 3 thru
 2 \ open poetry.blk to bring all the verses into play.
 3 \ Execute   LIMERICKS   to display all the verses constructed
 4 \ by this program.
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

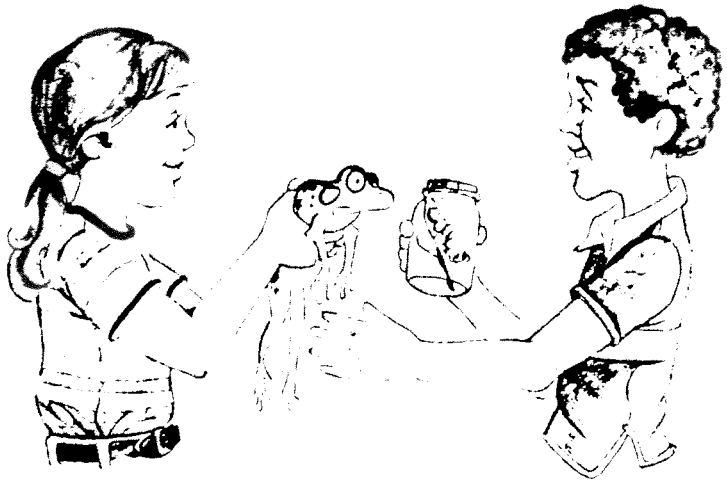
```
      2                                                        0
 0 \ records                                            16aug87cht
 1 variable seed   here seed !
 2 : random   seed @ 31421 *   6927 + dup seed ! ;
 3 : choose   random um* nip ;
 4 variable verses   1082 verses !
 5 : char ( addr -- )
 6    1024 /mod   block + c@ emit ;
 7 : .verse ( n -- )
 8    12 *   12 0 do   dup char   1+ loop   drop ;
 9 : poem ( print a randomly pick 4 verse poem)
10    cr cr
11    4 0 do   verses @ choose .verse   loop ;
12 : window.verse ( n -- )
13    12 * 2+ 2 spaces  10 0 do   dup char   1+ loop drop
14    2 spaces ;
15
```

```
      3                                                        0
 0 \ window dispaly                                     16aug87cht
 1 hex
 2 : clear ( x y -- )
 3    at   9 0 do  al emit bd emit ( sqaure fill) loop ;
 4 decimal
 5 : box   6 0 do   2dup clear   1+ loop   2drop ;
 6 : window   62 choose 17 choose   2dup box
 7    swap 2+ swap
 8    4 0 do   1+ 2dup at   verses @ choose window.verse   loop
 9    2drop ;
10 : limericks
11    dark   begin   window 1000 ms   key? until ;
12
13
14
15
```

```
      4
 0 \ print everything                                   23aug87cht
 1 2 3 thru
 2 : everything
 3    verses @ 0 do   cr cr
 4       1 dup 4 .r   4 0 do   dup window.verse 1+ loop   drop
 5    4 +loop ;
```

Listing 18. Chinese limericks

SWAP



XIX. THE SIMPLEST LINE DRAWING ROUTINE

Phil Koopman(1) published a very nice article on the Bresenham's line-drawing algorithm(2). This algorithm is the basis of most computer graphics packages due to its efficiency. It requires only 16 bit integer addition, subtraction, and multiplication by 2. The thing which troubled me was that it takes about 8 screens of source code to implement it. I did it a couple of years ago, using about the same amount of code as Phil's. The reason is that the program must include 4 slightly different routines to handle lines with slope in four different regions: 0 to 1, 1 to infinity, 0 to -1, and -1 to minus infinity. It is not possible to write a single routine to handle all four cases.

It troubles me because I feel intuitively that such a simple task--drawing a straight line--does not warrant 8 screens of code, especially in Forth. Any two year old kid knows how to do it on a newly painted wall. As we've seen editors in 3 screens, assemblers in 3 screens, data base management in 3 screens, and floating point package in one screen, 8 screens are really too much Tto draw straight lines.

In the summer of 1985, while I was studying recursive techniques, I found a very simple method to do line drawing. The code is very short, less than half of a screen. It was published in the FigAI Notes(3). Probably because of its brevity, this routine escaped attention. I thus feel a longer paper is necessary to give it some more credit which it deserves.

The code published in the FigAI Notes was written in F83X for Apple II. It is rewritten for IBM PC in F83, with a few more words to plot dots in graphic mode. The code is improved so that the testing of end of recursion is more accurate and truncation is replaced by rounding in the 2/ operation.

The algorithm is extremely simple. To draw a straight line between two points, you first find the middle point and plot it. This line is then broken into two equal segments. You then find the middle points of the two segments and plot them. And so forth till all points on the line are plotted. Recursion is really handy.

If the middle point is plotted before the line is segmented, the line is drawn by filling the points along the line. If we draw the end point at the end of a recursion tree, as shown in the code, the line is drawn backwards; i. e., from the end (x2,y2) to the starting point (x1,y1). To draw the line from (x1,y1) to (x2,y2), we must plot the starting point of a segment at the end of a recursion tree. This routine is thus

also suitable for plotters, not only for raster scanning devices.

The code as shown in Listing 19 is not as fast as Bresenham's algorithm, but there are lots of room for improvements, especially in the tests to determine the end of recursion. It could be implemented in machine code if speed is needed.

You have to excuse me for boasting a little bit, but this is my proudest discovery. I am pretty sure that some mathematician had already published it in some obscure journal long time ago. I sure like to put my name before this algorithm if nobody else claimed it.

Most recently, I did a little bit of research to see if this method was ever mentioned in the literature. I dug out an interesting treatise "Fundamental Algorithm for Computer Graphics"(4), in which the first section is devoted to line and area drawing algorithms. Bresenham is prominently present in this book. However, there is nothing about drawing straight lines using this recursive method. If the scholarship of this book can be relied upon, I felt that my chance of claiming the discovery is fairly good.

REFERENCES

1. Koopman, Phil. Jr.,The Bresenham Line-Drawing Algorithm, Forth Dimensions, Vol. 8, No. 9, pp. 12, 1987.

2. Bresenham, J. E., Algorithm for Computer Control fo a Digital Plotter, IBM Systems Journal, Vol. 4, No. 1, P.25-30, January 1965.

3. Ting, C. H., Studies of Recursion, in "FigAI Notes", Offete Enterprises, p. 58, 1985.

4. Earnshaw, R. A., Editor, Fundamental Algorithms for Computer Graphics, NATO ASI Series, Vol. 17, Springer-Verlag, Berlin, 1986.

```
      1
 0 \ Line drawing
 1 2 3 thru ( recursive line drawing)
 2 4 6 thru ( extended memory)
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
      2
 0 \ Video Services
 1 HEX
 2 CODE VIDEO ( cx dx ax -- )
 3    AX POP   DX POP   CX POP   10 INT   NEXT END-CODE
 4 : TEXT    0 0 2 VIDEO ;
 5 : GRAPH   0 0 4 VIDEO ;
 6 CODE PLOT ( x y color -- )
 7    AX POP   0C00 # AX ADD   DX POP   CX POP   10 INT
 8    NEXT END-CODE
 9 DECIMAL
10
11
12
13
14
15
```

```
      3
 0 \ The ultimate line drawing routine
 1 : draw ( x1 y1 x2 y2 -- )
 2    2over 2over   rot - abs >r
 3    - abs r> max   2 < if 2drop 3 plot exit then
 4    2over 2over   rot + 1+ 2/ >r ( y3 )
 5    + 1+ 2/ ( x3) r>   2dup 2rot   recurse   recurse ;
 6 : test1   640 0 do   0 0 i 400   draw   10 +loop ;
 7 : test2   400 0 do   0 0 640 i   draw   10 +loop ;
 8
 9
10
11
12
13
14
15
```

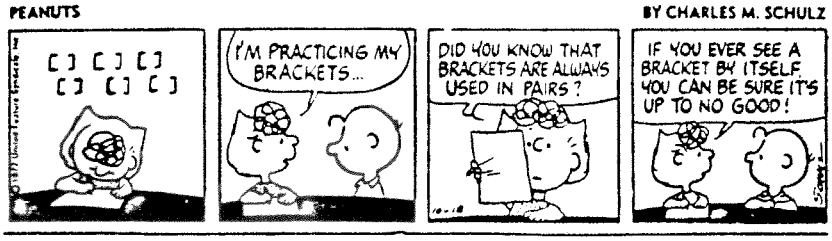
```
      6
16aug87cht \s                                          16aug87cht
```

```
      7
16aug87cht \ Video Services                            16aug87cht

          VIDEO   Call video service in IBM BIOS.

          TEXT    Return to text mode.
          GRAPH   Set to high resolution graphic mode.
          PLOT    Given a coordinate pair and a color code, paint
                  one dot on the screen.
```

```
      8
16aug87cht \ The ultimate line drawing routine          16aug87cht
          draw    Draw a straight line between (x1,y1) and (x2,y2).
                  Determine the end condition, where (x1,y1) and (x2,y2)
                  are within 1 pixel distance.
                  Find the mid point between (x1,y1) and (x2,y2).
                  Insert mid point between 1 and 2, then recurse twice
                  to draw the two segments.
          test1   Draw lines to fill lower left half of the screen.
          test2   Draw lines to fill upper right of the screen.
```

Listing 19. Recursive Line Drawing Routine

[ ]


PEANUTS
BY CHARLES M. SCHULZ
[ ] [ ] [ ]
[ ] [ ] [ ]
I'M PRACTICING MY BRACKETS...
DID YOU KNOW THAT BRACKETS ARE ALWAYS USED IN PAIRS?
IF YOU EVER SEE A BRACKET BY ITSELF, YOU CAN BE SURE IT'S UP TO NO GOOD!

XX. DIRTY WORDS IN FORTH

1. HISTORICAL BACKGROUND

In late 60's and early 70's, Forth matured when Chuck Moore put Forth on every computer within his reach. At that time, small computers used in observatories were minicomputers, most of them were 16 bit machines, imprinting Forth with the strong 16 bit flavor as reflected in Forth-77 standard. The width of numbers on the data and return stacks were 16 bits. However, addresses were generally what were natural for the specific minicomputer, mostly pointing to words of 16 bit quantities. For machines which used bytes as the basic storage unit, he would used byte addresses.

When figForth was implemented, the model provided by Bill Ragsdale assumed a host computer which used byte addresses exclusively, because the 8 bit microprocessors had become the dominant species in personal computers. FigForth addressed the needs of the users of these small computers. The side effect was that it legitimized byte addressing as the only way to address memory. This side effect was fossilized into the Forth-78 Standard and the subsequent Forth-79 and Forth-83 Standards, in the form of a set of standard words which explicitly assume byte addresses as their parameters.

Using 16 bit addresses for memory in the units of bytes, the maximum amount of addressable memory is 64 Kbytes. This memory barrier was broken several years ago, as microprocessors using 32 bit registers became available. If Forth were to become a significant player on advanced microcomputers, it would have to be able to use 32 bit numbers and addresses in a more natural fashion.

2. MEMORY SPACE AND ADDRESSES

The Forth standards assumes that addresses are pointers pointing to memory in increments of bytes. They are thus not compatible with machines which address memory in units other than bytes. There are indeed many minicomputers which address memory in 16 bit, 24 bit, 32 bit, and other odd units. Programs written according to any of the Forth standards cannot be ported to these machines without significant modifications. This inconsistency became very irritating with the Novix NC4000 chip, which is truely a Forth machine but imcompatible with the existing standards due to its 16 bit cell size.

The concept of memory space is important in understanding this problem. The same physical memory can be addresses by many different methods. We are conditioned to think that memory are always organized in bytes and addresses are pointers to bytes in memory. The byte memory space is only one among many methods to access memory. The bit memory space is important in graphics applications. Since Forth words are generally represented by 16 bit pointers, the 16 bit cell memory space is useful in organizing Forth dictionary and data arrays. For double integers and single precision floating point numbers, 32 bit memory space is appropriate. In one machine, there are simple operations to map one memory space to another. However, in machines which use different memory pointers to organize their memory, the mapping is not necessary straightforward.

We must acknowledge the existence of machines which do not use byte memory space and find ways to relax the standard so that these machines can be accommodated.

3. DEFICIENCY IN figFORTH, FORTH-79 AND FORTH-83

The most serious deficiency in these standards is the assumptions that memory is organized in bytes, and numbers and addresses are 16 bit in width. These assumptions must be relaxed in future standard for 32 bit machines. One has to realize that an advanced CPU can access memory for many different types of quantities, bytes or characters are only one type of quantities. Other quantities include 16 bit words, 32 bit long words, etc., and sometimes even individual bits. The standard should not demand that an address must always be a byte address. The address should be the one which points to memory in units most convenient for the machine, which may be bytes, 16 bit words, 32 bit long words, or whatever.

The width of the address and number should also be the most natural one for the machine, 16 bits at the least, but can be 24 bits, 32 bits or whatever.

It seems that a new standard with such relaxed requirements will be difficult to formulate and to be agreed upon. However, if we look carefully at the Forth Standards, it will be obvious that only a small portion of the word set needs to be clarified. These words are characterized by that they require addresses explicitly refer to bytes in memory. If they are excluded from the standard or re-worded so that they do not require byte addresses, a more encompassing standard can be arrived at which can serve the needs of the Forth community for a long time.

4. THE DIRTY WORDS IN FORTH

Most words in the standards are 'pure' words, which are not sensitive to the size of the memory cell in a computer. They assume that the widths of the affected parameters are that of the items on the stacks. They include stack words, arithmetic and logical operations, and many others. In a 16 bit machine, they use numbers or addresses of 16 bit width. In a 32 bit machine, they use numbers or addresses of 32 bit width.

The 'dirty' words are those which use byte addresses explicitly to access memory and those which assumes that the parameters involved are 16 bit numbers. The dirty words cannot be ported from a byte addressing machine to a cell addressing machine without changing the program. Problems will also surface in porting these words from a 16 bit machine to a 32 bit machine.

The 'pure' words and the 'dirty' words in the Forth-83 Standard are grouped and listed in Figure 37 for comparison. It is very encouraging that only a very small portion of the standard words are actually dirty. Hence it should not be too much trouble in cleansing the standard to become a pure one.

All the dirty words uses byte addresses to get information from memory or store data in bytes to memory. The addresses are strictly in the byte memory space and hence cannot be used in a machine operated in the cell memory space. The byte space must be mapped or translated into cell memory space for the desired function.

5. SUGGESTIONS FOR A MORE POWERFUL FORTH STANDARD

To design a more powerful and more encompassing Forth standard, the following suggestions seem appropriate:

a. The language in the standard must be relaxed and does not requiring that numbers and address are 16 bits in width.
b. The widths of numbers and addresses should be the 'natrual width' of the computer, though not less than 16 bits.
c. The 'dirty' byte addressing words must be isolated into an extension layer, but still available to be used on a byte addressible computer.
d. Many 'dirty' words like CMOVE, FILL, etc, should have equivalent words in the cell space to perform equivalent function more efficiently.

Figure 37. Pure and Dirty words in Forth-83 Standard

PURE WORDS IN FORTH-83 (118 out of 134 words)

```
Logic:       NOT  AND  OR  XOR
Stack:       DUP  DROP  SWAP  OVER  ROT  PICK  ROLL
             ?DUP  >R  R>  R@  DEPTH
Comparison:  <  >  =  0<  0>  0=  U<
Arithmetic:  +  -  1+  1-  2+  2-  *  UM*  UM/MOD
             2/  /  MOD  /MOD  */MOD  */
             MAX  MIN  ABS  NEGATE
             D<  D+  DNEGATE
Memory:      @  !  +!
Numeric:     BASE  DECIMAL  <#  #  #S  HOLD  SIGN
Control:     DO  LOOP  +LOOP  LEAVE  I  J  IF  ELSE
             THEN  BEGIN  UNTIL  WHILE  REPEAT
             EXIT  EXECUTE
Terminal:    .  U.  ."  .(  CR  EMIT  SPACE  SPACES
             KEY
Storage:     BLOCK  BUFFER  UPDATE  SAVE-BUFFERS  FLUSH
             BLK
Program:     FORTH-83  QUIT  ABORT  ABORT"
Dictionary:  HERE  PAD  TIB  >BODY
Compiler:    LOAD  (  ,  DOES>  [COMPILE]  IMMEDIATE
             COMPILE  STATE  LITERAL  [  ]  #TIB
Vocabulary:  FORTH  DEFINITIONS  '  [']  FORGET
Defining:    :  ;  CREATE  VARIABLE  CONSTANT  VOCABULARY
```

DIRTY WORDS IN FORTH-83 (16 out of 134 words)

```
Memory:      C@  C!  CMOVE  CMOVE>  FILL  COUNT
             -TRAILING
Numeric:     CONVERT  #>
Terminal:    TYPE  EXPECT  SPAN
Compiler:    ALLOT  WORD  >IN
Vocabulary:  FIND
```

XXI. STATISTICS OF WORD USAGE IN F83

1. INTRODUCTION

How often are various Forth words used is a question interesting to most Forth programmer because this type of information can lead to better design and the optimization of Forth systems. Most often used words should be coded in machine language for execution speed. They should also be at the top of the dictionary to minimize the time for interpretation and compilation. A number of year ago, before Don Colburn made his first million, he mentioned in a FORML meeting at Hayward, California that he used an extra cell in the word header to accumulate statistics of word usage, either during compilation or during execution. He also mentioned that the most often used Forth word was (for comments, which was rather unexpected at that time. Since I did not have the luxury to metacompile my own system with that type of flexibility at the time, this concept had remained to be a distant curiosity for me.

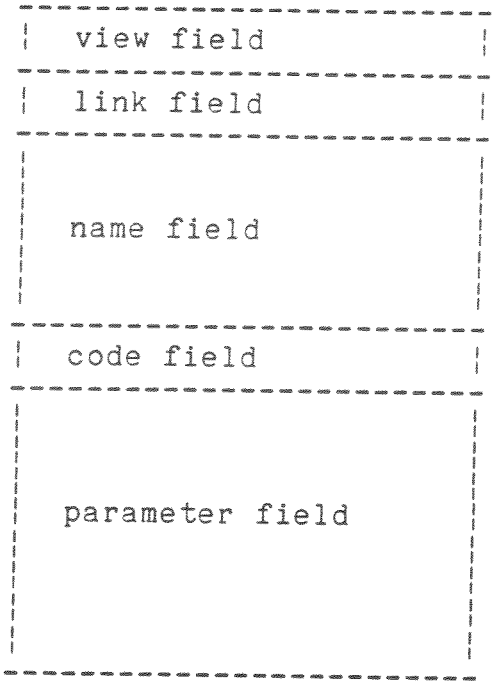


Figure 38. Forth word layout in F83.

After plunging myself into the F83 system produced by Mike Perry and Henry Laxen, I found a ready solution to analyze the Forth word usage without much hard work. The secret is

in the extra cell used in F83 to store the view file information. As shown in Figure 38, the Forth words in F83 are laid out in the dictionary composed of 5 fields: the view field, the link field, the name field, the code field, and the parameter field.

The view field stores a file number in it upper 4 bit subfield and a block number in the lower 12 bit subfield, allowing the source screen containing the word definitions to be retrieved from the disk and viewed by the user. If I am not going to used the view field for viewing purposes, I am free to use it for whatever purpose I choose to do with it. Why not use it to accumulate the statistics of Forth word usage?

To use the view field to to statistical analysis, I must do it in the following sequence:

a. Clear the view fields of all words in the dictionary.
b. Build a 'word' processor which will scan screens of code and increment the view field when the corresponding word is encountered in a screen.
c. Run it through as many source files as available.
d. Tabulate the statistics.

2. THE IMPLEMENTATION

The program shown in Listing 20 performs the above functions. The program looks extremely simple because it utilizes many powerful and interesting features in the F83 system, which requires some explanation.

The most important feature I used is the vectored execution procedure, which allows me to assign a number of different tasks to a single word. For example, I want to scan the dictionary and clear all the view fields before analyzing word usage. After the statistics is collected, I want to scan the dictionary and print the contents of the view fields. The scanning operations in both cases are identical. The difference is the actions I have to take after I find a view field. Anticipating that different actions are to be taken, I defined a vectored word WORK as a DEFERred word and use it in the definition of the scanning word WORKS, which follows the dictionary link to locate every word in the dictionary and perform WORK on each of them.

WORKS is complicated by the fact that F83 hashes the dictionary linkage into four threads, and all four threads have to be scanned when traversing the dictionary. The definition is very similar to the word DEFINED, which does the dictionary search for the text interpreter in F83. It scans all four threads and processes the one with the highest link field address. The process continues until all four link addresses are reduced to zero, indicating the end of the threads. The scanning is performed only on the FORTH vocabulary, in which all the primitive Forth words reside. Usage of words in other vocabulary are much less frequent and the statistics is less significant than

those word in the FORTH vocabulary.

After INIT-VIEW is defined to clear the view field, given a link field address, we can zero the view field of all the words in the FORTH vocabulary by vectoring WORK to INIT-VIEW and execute WORKS. After the statistics is accumulated in all the view fields, we will vector WORK to PRINT-VIEW and execute WORKS. This time around, WORKS will print the contents of the view fields with the corresponding word names.

ACCUMULATE in screen 19 is the 'word' processor which processes source screens very much like what INTERPRET would do. If a word is found in the dictionary by DEFINED, the view field of this word is incremented. If a word is not found in the dictionary, actually the FORTH vocabulary, it is simply skipped over. I couldn't care less if it is a number, which will be ignored also, except 0, 1, 2 and 3, which are Forth words.

In F83, LOAD is also a vectored word. I define [LOAD] to use ACCUMULATE to analyze the contents of a screen. After LOAD is vectored to [LOAD], LOADing a screen will accumulate counts to words which appear in this screen. THRU can be used to analyze a range of screens in a file. Running a large number of screens through, we can get fairly representative statistics for all the commonly used Forth words.

F83 is a very large system consists of many system and utility programs. It serves very well as a data base for the purpose of statistical analysis. Using the above technique, I ran all the source screens through this word processor, including 7 F83 source files with 230 screens of code. There are 555 words in its FORTH vocabulary and total occurrence of these words is 10603. The result is tabulated in Figure 39. The most often occurred words, those counted 30 times or more, are listed in Figure 40.

The most obvious application derivable from the above analysis is that if we can arrange the dictionary so that those most often used words are on the very top of the dictionary, the speed of interpretation and compilation will be significantly improved because the dictionary searching time can be reduced. Another interesting observation is that comments were heavily used in the F83 system using (S , \ , and (. This is of course dictated by good programming style and in-line documentation.

2	EMPTY	3	(CONVEY)	2	DOES-SIZE	79	VARIABLE	4	(")	1	UD.	8	B/FCB	14	BLK	6	2OVER	10	?DUP
2	MARK	2	.TO	3	DOES-OP	69	CONSTANT	24	[COMPILE]	3	(UD.)	3	REC/BLK	8	VOC-LINK	10	2SWAP	2	FLIP
3	HELLO	2	HOP	28	LABEL	2	RECURSIVE	27	[']	13	.R	3	B/REC	8	WIDTH	62	2DUP	19	-ROT
1	BACKGROUND:	3	CONVEY-COPY	2	UTILITY.BLK	712	:	113	'	17	.	12	B/BUF	4	'TIB	38	2DROP	41	ROT
1	ACTIVATE	3	U/D	2	CPU8086.BLK	711	;	12	?MISSING	3	(.)	11	#BUFFERS	26	CONTEXT	15	2!	24	NIP
3	SET-TASK	5	HOPPED	2	EXTEND86.BLK	65	]	2	CRASH	3	U.R	5	QUERY	9	#VOCS	18	2@	7	TUCK
2	TASK:	1	VIEW	2	KERNEL86.BLK	66	[	5	CONTROL	7	U.	4	TIB	11	CURRENT	2	WITHIN	100	OVER
1	RESUME	3	@VIEW	6	VIEWS	41	DOES>	35	ASCII	3	(U.)	3	EXPECT	3	CSP	5	BETWEEN	153	SWAP
3	DEBUG	2	COPY	5	VIEW-FILES	1	:CODE	4	DLITERAL	2	OCTAL	2	CC-FORTH	10	LAST	2	MAX	242	DUP
1	LISTING	5	(COPY)	2	SAVE-SYSTEM	5	(;CODE)	20	LITERAL	14	DECIMAL	3	CC	5	R#	8	MIN	109	DROP
5	SHOW	2	ESTABLISH	6	FROM	8	:USES	48	IMMEDIATE	13	HEX	3	DEL-IN	6	DPL	11	>	2	RP!
2	(SEMIT)	3	L	3	OPEN	50	ASSEMBLER	22	COMPILE	5	#S	31	CHAR	6	WARNING	11	<	1	RP@
2	(PAGE)	2	B	1	DEFINE	4	(;USES)	7	EVEN	68	#	2	(CHAR)	10	STATE	5	U>	2	SP!
2	FORM-FEED	10	N	1	B:	6	REVEAL	8	ALIGN	3	SIGN	2	CR-IN	4	PRIOR	13	U<	7	SP@
5	PAGE	1	::	1	A:	5	HIDE	57	C,	8	#>	2	P-IN	14	SCR	2	?NEGATE	7	CMOVE>
4	#PAGE	1	MANY	1	DRIVE?	5	?CSP	103	,	8	<#	2	RES-IN	28	EMIT	7	<>	26	CMOVE
3	LOGO	1	TIMES	1	DIR	6	!CSP	28	ALLOT	7	HOLD	3	BACK-UP	9	PRINTING	47	=	18	C!
2	L/PAGE	5	#TIMES	1	CREATE-FILE	53	CREATE	5	INTERPRET	12	NUMBER	2	(DEL-IN)	11	IN-FILE	17	0<>	58	C@
5	INIT-PR	5	LARGEST	9	ROOT	2	"CREATE	4	STATUS	2	(NUMBER)	2	BS-IN	21	FILE	5	0>	145	'
4	EPSON	1	IND	3	-->	22	.VIEW	3	?STACK	3	NUMBER?	1	BEEP	15	HLD	14	0<	301	@
2	SEE	3	INDEX	16	+THRU	100	WHILE	8	DEFINED	2	(NUMBER?)	5	BACKSPACE	14	BASE	40	0=	3	(?LEAV
5	(SEE)	3	.LINE0	5	THRU	225	ELSE	4	?UPPERCASE	2	CONVERT	22	SPACES	5	OFFSET	7	UM/MOD	3	(LEAVE
3	ASSOCIATIVE:	2	TRIAD	7	?ENOUGH	22	IF	3	FIND	4	DOUBLE?	33	SPACE	8	#LINE	1	U*D	2	J
3	CASE:	4	LIST	9	?	8	REPEAT	14	#THREADS	4	DIGIT	28	TYPE	15	#OUT	7	UM*	50	I
2	MAP	3	.SCR	726	(S	20	AGAIN	3	(FIND)	25	LOAD	3	CRLF	9	DP	21	2-	14	PAUSE
6	OUT	4	?CR	'15	\	5	UNTIL	8	HASH	3	(LOAD)	3	(EMIT)	6	RPO	25	1-	241	NOOP
5	DL	3	?LINE	4	L/SCR	67	+LOOP	1	VIEW>	2	DEFAULT	3	(PRINT)	9	SPO	45	2+	1	GO
1	DU	3	RMARGIN	37	C/L	26	LOOP	3	>VIEW	3	VIEW#	2	PR-STAT	6	LINK	61	1+	7	PERFOR
5	DUMP	4	LMARGIN	1	RECURSE	46	?DO	2	>LINK	7	FLUSH	86	CR	9	ENTRY	5	B#	7	EXECUT
2	.HEAD	5	HIDDEN	0	B	223	DO	17	>NAME	4	SAVE-BUFFERS	8	KEY	4	TOS	1	U2/	14	>NEXT
2	?.A	1	0<=	2	0	40	THEN	30	>BODY	3	EMPTY-BUFFERS	13	KEY?	1	#/	3	2/	12	BOUNDS
3	?.N	3	0>=	0	DUMP	8	BEGIN	2	LINK>	3	IN-BLOCK	4	(CONSOLE)	2	#/MOD	37	2#	4	(?DO)
2	DLN	4	>=	19	.ID	7	?LEAVE	8	NAME>	12	BLOCK	3	(KEY)	5	MOD	39	3	4	(DO)
2	EMIT.	2	<=	4	.S	14	LEAVE	7	BODY>	3	(BLOCK)	5	(KEY?)	6	/	58	2	4	(+LOOP
3	D.2	2	U>=	4	DEPTH	8	?>RESOLVE	6	L>NAME	3	BUFFER	26	BDOS	10	/MOD	103	1	4	(LOOP)
2	.2	4	U<=	2	BYE	14	?>RESOLVE	1	N>LINK	3	(BUFFER)	2	COMPARE	26	*	241	0	6	?BRANC
6	A	1	MS	3	START	14	?>MARK	3	FORTH-83	2	MISSING	2	CAPS-COMP	2	MU/MOD	30	+!	6	BRANCH
8	SHADOW	3	FUDGE	2	OK	2	<RESOLVE	3	DONE?	1	DISCARD	3	COMP	3	M/MOD	6	ABS	6	(LIT)
3	(WHERE)	2	P!	4	INITIAL	2	<MARK	11	TRAVERSE	4	UPDATE	7	-TRAILING	2	*D	70	-	17	UP
1	FIX	8	PC!	3	COLD	2	>RESOLVE	250	\S	2	ABSENT?	17	PAD	1	DMAX	8	NEGATE	6	UNNEST
3	EDIT	2	P@	3	WARM	2	>MARK	29	(	2	LATEST?	73	HERE	1	DMIN	140	+	15	EXIT
2	ED	2	PC@	5	BOOT	9	?CONDITION	6	.(	4	CAPACITY	3	UPPER	2	D>	54	OFF	80	FORTH
4	DONE	1	MULTI	8	QUIT	4	ABORT	11	>TYPE	32	DOS	2	UPC	3	D<	27	ON		
15	EDITOR	1	SINGLE	6	RUN	25	ABORT"	6	WORD	13	SWITCH	3	MOVE	4	DU<	1	CTOGGLE		
5	DARK	1	STOP	65	IS	4	(ABORT")	6	'WORD	4	FILE?	1	LENGTH	4	D=	1	CRESET		
10	AT	3	WAKE	3	(IS)	2	(?ERROR)	2	PARSE	4	.FILE	28	COUNT	6	DO=	7	CSET		
4	-LINE	3	SLEEP	6	>IS	3	?ERROR	4	PARSE-WORD	5	WRITE-BLOCK	10	BLANK	2	?DNEGATE	18	FALSE		
5	BLOT	4	!LINK	16	USER	5	WHERE	2	SOURCE	4	READ-BLOCK	14	ERASE	5	D-	30	TRUE		
2	REPLACE	3	@LINK	6	#USER	2	FORGET	9	(SOURCE)	5	>UPDATE	4	FILL	1	D2/	38	NOT		
5	INSERT	8	LOCAL	116	CODE	3	(FORGET)	5	PLACE	14	BUFFER#	8	CAPS	1	D2*	11	XOR		
8	DELETE	3	INT#	3	AVOC	2	TRIM	4	/STRING	5	>END	4	BELL	3	DABS	51	OR		
7	SEARCH	2	RESTART	4	2VARIABLE	4	FENCE	2	SCAN	9	>BUFFERS	5	BS	2	S>D	60	AND		
2	SCAN-1ST	2	(PAUSE)	2	2CONSTANT	61	"	1	SKIP	3	INIT-RO	22	BL	5	DNEGATE	3	ROLL		
4	FOUND	1	UNBUG	91	DEFINITIONS	67	."	1	D.R	8	FIRST	5	END?	5	D+	8	PICK		
15	TO	5	BUG	21	VOCABULARY	4	,"	3	D.	4	>SIZE	4	#TIB	1	2ROT	21	R@		
4	CONVEY	3	DOES?	35	DEFER	4	(.")	1	(D.)	4	LIMIT	6	SPAN	3	4DUP	50	>R		
									UD.R	10	DISK-ERROR	28	>IN	4	3DUP	59	R>		

Figure 39. Statistics of FORTH Word Usage

Figure 40. Most often occurred Forth words in F83.

(S	726	CR	86	OR	51
;	712	FORTH	80	ASSEMBLER	50
:	711	VARIABLE	79	>R	50
\	475	HERE	73	I	50
@	301	-	70	IMMEDIATE	48
(	250	CONSTANT	69	=	47
DUP	243	#	68	DO	46
0	241	LOOP	67	2+	45
NOOP	241	."	67	ROT	41
IF	225	[	66	0=	40
THEN	223	]	65	BEGIN	40
SWAP	153	IS	65	3	39
!	145	2DUP	62	NOT	38
+	140	1+	61	2DROP	38
CODE	116	"	61	2*	37
?MISSING	113	AND	60	DEFER	35
DROP	109	R>	59	ASCII	35
1	103	C@	58	SPACE	33
,	103	2	58	DOS	32
OVER	100	C,	57	CHAR	31
ELSE	100	OFF	54	>BODY	30
DEFINITIONS	91	CREATE	53	+!	30
				TRUE	30

```
                ok
18 LIST 19 LIST
Scr # 18          B:META86.BLK
  0 \ Statistical analysis of words                              25JAN85CHT
  1 DEFER WORK (S link --- , to do misc. works on vocabualry words)
  2 ' NOOP IS WORK
  3 : WORKS   (S -- , scan vocabulary and WORK on each word)
  4    CONTEXT @ HERE #THREADS 2* CMOVE
  5    BEGIN   HERE #THREADS LARGEST   DUP
  6    WHILE   DUP WORK   @ SWAP !   REPEAT   2DROP   ;
  7 : INIT-VIEW   (S link --- , clear a word counter.)
  8    2- OFF   ;
  9 : PRINT-VIEW   (S link --- , print ontents of a word counter.)
 10    CR   DUP 2- @ 6 .R   3 SPACES   L>NAME .ID   ;
 11 EXIT
 12 ' INIT-VIEW IS WORK   WORKS   ( Initialize all word counters)
 13 ' PRINT-VIEW IS WORK  WORKS   ( Print all word counters)
 14
 15

Scr # 19          B:META86.BLK
  0 \ New load for statistics                                    23JAN85CHT
  1 : ACCUMULATE  (S --- , text interpreter to increment  counters)
  2    BEGIN   DEFINED   IF >VIEW 1 SWAP +!   ELSE DROP THEN
  3    FALSE DONE? UNTIL   ;
  4 : [LOAD]     (S n -- , interpret block n, like LOAD )
  5    FILE @ >R   BLK @ >R   >IN @ >R
  6    64 >IN ! ( Skip 0th line to avoid wrap-around.)
  7    BLK ! IN-FILE @ FILE ! ACCUMULATE R> >IN ! R> BLK !
  8    R> !FILES   ;
  9 EXIT
 10 ' [LOAD] IS LOAD   ( Use [LOAD] to do the WORKS)
 11 CAPS OFF           ( Do case sensitive compare and counting.)
 12 OPEN <file>        ( Select a source file to analyze.)
 13 1 10 THRU          ( Accumulate word statistics.)
 14   ...              ( Repeat for all source files.)
 15 ' PRINT-VIEW IS WORK   WORKS   ( Print results.)
 ok
```

Listing 20. Statistical analysis of words

XXII. ZAPPING THE F83 DICTIONARY

F83 is a very large system and there are about 1000 words defined in it. Some efforts were exercised to organize this huge amount of words by grouping them into 9 different vocabularies. However, the vocabularies are not well balanced because the FORTH vocabulary is considerably larger than all the other vocabularies. Many vocabularies contain fewer than 10 words. The distribution of words in the F83/8086/PCDOS is shown in Figure 41:

Figure 41. Word distribution in F83/8086

Vocabulary	Number of words
FORTH	555
ASSEMBLER	219
EDITOR	80
DOS	36
HIDDEN	35
BUG	14
ROOT	9
SHADOW	9
USER	4

The FORTH vocabulary is really too big to be useful. This is especially evident when one uses the command WORDS to list this vocabulary. It is very difficult to make some sense of the huge list of words scrolling over the CRT screen. It is downright frightening to a new user as he struggles through the F83 system. To be more friendly and useful, the FORTH vocabulary should be less than a screenful so that it can be listed and inspected conveniently. Reducing the size of the FORTH vocabulary may also improve the dictionary searching during compilation and interpretation.

The FORTH vocabulary can be reduced by carefully specifying the proper current vocabulary during the metacompilation. It requires lots of changes in the KERNEL and the UTILITY source files. A much easier way is to relink words into the appropriate vocabularies. Here I propose the new word ZAP as defined in Screen 90 of Listing 21. ZAP removes a word from the context vocabulary and relinks it to the top of the current vocabulary. Using ZAP, we can move any word from any vocabulary to any other vocabulary, or change the linking order of words within one vocabulary.

ZAP is defined using SEARCH and RELINK. SEARCH searches through the context vocabulary for the next word and returns two link field addresses (lfa): the lfa on the top of stack is that of the word searched, and the next lfa is that of the word prior to the searched word in the same linked thread. RELINK copies the contents of the second lfa into the cell pointed to by the first lfa, and thus removes the searched word from the context vocabulary. RELINK then links the searched word to the end of the current vocabulary by changing the contents of the variable CURRENT and the link field of the searched word.

This simple operation logically excises one word from the context vocabulary and adds it to the current vocabulary. Physically, the word stays at the same memory location. Only its link field is modified. After a few of these operations, the linkage in the context and current vocabularies will become randomized. It will be very difficult to forget words without crashing the system. Nevertheless, this zapping process is designed to rearrange the vocabularies below the FENCE, where words cannot be forgotten.

ZAPPING puts ZAP in a loop so that we can zap a large number of words by loading one or more screens as shown in Screens 91 to 93. This way the user can customize a F83 system very easily by collecting all the words he wants to remove from one vocabulary to another in a small number of screens and load them when needed.

The results of ZAPPING is shown in Figure 42, in which a much leaner and cleaner FORTH vocabulary is listed. This list fits snugly inside one CRT screen. It contains all the Forth-83 standard words and the utility words which I use very often during normal programming over the years. I think this is a minimum collection of Forth words needed by a F83 user doing typical program development work. It is also a better list of words to be presented to new users for them to learn the F83 system and to polish their Forth skill.

A smaller FORTH vocabulary will certainly reduce the compilation time because FIND does not have to traverse through the long FORTH vocabulary. However, I have not done accurate timing to determine the saving in compilation time. Another application of ZAP is to rearrange the search order in a single vocabulary. Linking the most often used words to the top of the FORTH vocabulary would also reduce the compilation time. The process to change the searching order is to ZAP all the words in a vocabulary and then ZAP them back in the reversed order as desired. The desired order of words in the FORTH vocabulary is probably that according to the statistics of word usages discussed in the previous paper.

```
ZAPPING  DOCUM.BLK  DEBUG  SHOW  SEE  DL  DU  DUMP  A  SHADOW
FIX  EDIT  ED  DONE  EDITOR  TO  CONVEY  VIEW  COPY  L  B
N  WORDS  INDEX  TRIAD  LIST  HIDDEN  P!  PC!  P@  PC@  BUG
LABEL  FROM  OPEN  B:  A:  DIR  CREATE-FILE  MORE  ROOT  THRU
?  (S  \  B  @  DUMP  .S  DEPTH  BYE  QUIT  CODE
DEFINITIONS  VOCABULARY  VARIABLE  CONSTANT  ;  :  ]  [  DOES>
;CODE  ASSEMBLER  CREATE  WHILE  ELSE  IF  REPEAT  AGAIN
UNTIL  +LOOP  LOOP  DO  THEN  BEGIN  LEAVE  ABORT  ABORT"
FORGET  "  ."  ,"  [COMPILE]  [']  '  ASCII  IMMEDIATE
COMPILE  C,  ,  ALLOT  INTERPRET  FIND  >BODY  \S  (  .(
WORD  D.R  D.  UD.  .R  .  U.R  U.  OCTAL  DECIMAL  HEX  #S
#  SIGN  #>  <#  HOLD  NUMBER  CONVERT  LOAD  FLUSH
SAVE-BUFFERS  EMPTY-BUFFERS  BLOCK  BUFFER  UPDATE  DOS  QUERY
TIB  EXPECT  SPACES  SPACE  TYPE  CR  KEY  KEY?  -TRAILING
PAD  HERE  MOVE  LENGTH  COUNT  BLANK  ERASE  FILL  SPAN  >IN
BLK  WIDTH  CONTEXT  CURRENT  WARNING  EMIT  PRINTING  BASE
*/  */MOD  MOD  /  /MOD  *  MU/MOD  M/MOD  *D  D>  D<  D=
D0=  D-  D2/  D2*  DABS  S>D  DNEGATE  D+  2OVER  2SWAP
2DUP  2DROP  2!  2@  WITHIN  BETWEEN  MAX  MIN  >  <  U>
U<  =  0>  0<  0=  UM/MOD  U*D  UM*  2-  1-  2+  1+  U2/
2/  2*  3  2  1  0  +!  ABS  -  NEGATE  +  NOT  XOR  OR
AND  ROLL  PICK  R@  >R  R>  ?DUP  -ROT  ROT  OVER  SWAP
DUP  DROP  CMOVE>  CMOVE  C!  C@  !  @  J  I  EXECUTE  EXIT
FORTH
```

Figure 42. Zapped FORTH vocabulary

```
    90                                                  03sep86cht
0 \ Zapping unused words
1 : SEARCH ( -- lfa1 lfa2 )
2    32 WORD   DUP CONTEXT @ HASH   DUP @ ( string lfa1 lfa2 )
3    BEGIN   DUP
4    WHILE
5       DUP 3 + ( nfa ) 3 PICK COUNT     2DUP + 1- 128 SWAP CSET
6       COMPARE   IF   NIP DUP @
7       ELSE   ROT DROP EXIT   THEN
8    REPEAT   2DROP ?MISSING   ;
9 : RELINK ( lfa1 lfa2 -- )
10    DUP @ ROT ! ( delete from context vocabulary )
11    HERE CURRENT @ HASH   2DUP @ SWAP   ! ( new link )
12    ! ( current vocabulary thread )     ;
13 : ZAP ( -- )   SEARCH RELINK ;
14 : ZAPPING   0 DO ZAP LOOP ;
15
```

```
    91                                                  02SEP86CHT
 0 \ Zapping
 1 HIDDEN DEFINITIONS FORTH
 2 118 ZAPPING
 3 EMPTY MARK HELLO BACKGROUND: ACTIVATE SET-TASK TASK: (SEMIT)
 4 (PAGE) FORM-FEED #PAGE LOGO L/PAGE   (SEE) ASSOCIATIVE: CASE:
 5 MAP OUT .HEAD ?.A ?.N DLN EMIT. D.2   .2 (WHERE) REPLACE INSERT
 6 DELETE SEARCH SEARCH SCAN-1ST FOUND   (CONVEY) .TO CONVEY-COPY
 7 U/D @VIEW (COPY) ESTABLISH :: MANY   TIMES #TIMES LARGEST .LINE0
 8 .SCR ?CR ?LINE RMARGIN LMARGIN MULTI SINGLE STOP WAKE SLEEP
 9 !LINK @LINK LOCAL INT# RESTART (PAUSE) UNBUG DOES? DOES-SIZE
10 DOES-OP VIEWS VIEW-FILES DEFINE DRIVE? ?ENOUGH L/SCR C/L .ID
11 START INITIAL RUN IS (IS) >IS #USER AVOC (;CODE) ;USES (;USES)
12 REVEAL HIDE ?CSP !CSP "CREATE ,VIEW ?<RESOLVE ?<MARK ?>RESOLVE
13 ?>MARK <RESOLVE <MARK >RESOLVE >MARK ?CONDITION (ABORT")
14 (?ERROR) ?ERROR (FORGET) TRIM FENCE (.") (") ?MISSING CRASH
15 EVEN ALIGN STATUS ?STACK DEFINED ?UPPERCASE #THREADS (FIND)
```

```
    92                                                  03sep86cht
 0 \ More zapping
 1 117 ZAPPING
 2 HASH VIEW) >VIEW >LINK >NAME LINK>   NAME> BODY> L>NAME N>LINK
 3 DONE? TRAVERSE >TYPE 'WORD PARSE PARSE-WORD SOURCE (SOURCE)
 4 PLACE /STRING SCAN SKIP (D.) (UD.)   (.) (U.) (NUMBER?) DOUBLE?
 5 DIGIT (LOAD) DEFAULT VIEW# IN-BLOCK (BLOCK) (BUFFER) MISSING
 6 DISCARD ABSENT? LATEST? FILE? .FILE WRITE-BLOCK READ-BLOCK
 7 >UPDATE BUFFER# >END >BUFFERS INIT-R0 FIRST >SIZE LIMIT
 8 DISK-ERROR B/FCB REC/BLK B/REC B/BUF #BUFFERS CC-FORTH CC
 9 DEL-IN CHAR (CHAR) CR-IN P-IN RES-IN BACK-UP (DEL-IN) BS-IN
10 CRLF (EMIT) (PRINT) PR-STAT (CONSOLE) (KEY) (KEY?) BDOS
11 COMPARE CAPS-COMP COMP UPPER UPC CAPS BELL BS BL END? #TIB
12 'TIB #VOCS CSP LAST DPL PRIOR IN-FILE #LINE #OUT TOS RP! RP@
13 SP! SP@ (?LEAVE) (LEAVE) PAUSE NOOP GO PERFORM >NEXT (?DO)
14 (DO) (+LOOP) (LOOP) ?BRANCH BRANCH (LIT) UP UNNEST
15
```

```
    93                                                  26sep86cht
03sep86cht \ Zap more words
88 ZAPPING
ZAP RELINK RESUME LISTING FOOTING DARK AT -LINE BLOT
HOP HOPPED IND 0<= 0>= >= <= U>= U<= FUDGE --> +THRU
RECURSE COLD WARM BOOT   RECURSIVE ?DO ?LEAVE CONTROL
(NUMBER) NUMBER? SWITCH BEEP BACKSPACES VOC-LINK R#
DMAX DMIN DU< ?DNEGATE 2ROT 4DUP 3DUP ?NEGATE B# OFF ON
CTOGGLE CRESET CSET FALSE TRUE FLIP BOUNDS
PAGE INIT-PR EPSON MS
UTILITY.BLK CPU8086.BLK KERNEL86.BLK EXTEND86.BLK SAVE-SYSTEM
OK USER 2VARIABLE 2CONSTANT DEFER WHERE DLITERAL LITERAL
FORTH-83 UD.R CAPACITY STATE SCR FILE HLD OFFSET DP RP0 SP0
LINK ENTRY <> 0<> NIP TUCK
ONLY FORTH ALSO HIDDEN ALSO FORTH DEFINITIONS
```

Listing 21. Zapping unused words