RHB

the datac 1000 users' group

VOL. 2 NO. 1
MARCH 1978

Editor: John Prenis

Contributors this issue:

Carmen DiCamillo Edwin R. Morris
Rolland James
John Prenis

Subscription rates are $5 for six issues. Send subscription requests to Datac Engineering, P O Box 406, Southampton, Pa. 18966. Send letters, articles and news to John Prenis, 161 W. Penn St., Philadelphia, Pa. 19144. You can make us very happy by typing them single spaced in columns 13.5 cm. (5.3 in.) wide.

Once again I must apologize for a late issue. This time it was the need to finish the software for the cassette interface that caused the delay. I think you will find it worth the wait. A 4K program that takes 20 minutes to load using paper tape takes slightly more than a minute to load from cassette. That's fast!

Those of you who don't feel ready for a video terminal will be glad to hear that a fellow Datac user is working on a hex keyboard and display. It uses next to no hardware, and we hope to have the details in a forthcoming issue. Another interesting piece of hardware is the new memory board. In case you've forgotten, it has room for 16K of 2102's, a 2708, and a PROM programmer.

A slightly revised version of the DATAC 1000 board is now available. To start with, the stand-up resistors are gone from the switch register.

For those of you who are curious about the 6800 or who want to run 6800 software, the new board will accept either a 6802 or a 6502. A 6800 monitor is now being planned that will combine the best features of TIM and MIKBUG. With the addition of some jumpers the board will accomodate 2K 2716 ROMs in place of 2708's, giving 4K of ROM on board. Also 2114 RAM chips can be substituted for the 2111's, giving 4K of on board RAM. It probably wouldn't be too hard to add these features to the older boards as well. This is quite a board, folks.

Many thanks to those of you who sent articles and programs. Unfortunately lack of space prevents us from using any of them this time. Keep up the good work!

John Prenis

The Cassette Interface, Part II

by Carmen DiCamillo

In part one of this article we discussed the basic theory of the cassette interface and its hardware. Since then, we have made some minor hardware changes. Change the following jumpers on U-26:

Pin #3 from GND to +5V

Pin #4 from GND to +5V

Pin #22 from GND to +5V

The following is a complete list of U-26 jumpers:

Pin #3	+5V	Pin #9	GND	Pin #17	GND
4	+5V	10	GND	18	GND
5	GND	11	+5V	19	GND
6	GND	13	GND	20	+5V
7	GND	14	+5V	21	GND
8	GND	15	GND	22	+5V
		16	GND		

Making these jumper changes will adjust the cassette interface speed, to the clock speed divided by 28 (U-26), divided by 16 (ACIA divisor), rendering us the rate 2232 bits per second:

1.000 MHZ ÷ 28 ÷ 16 = 2232 bits/sec

We found it advisable to replace R42 with a 20K resistor. Finally, it is recommended that the PC trace coming from pin 14 of U-22 be cut and a diode be inserted (cathode toward the chip). This is to prevent a negative voltage being applied to this pin.

THE SOFTWARE

Table #1 is a listing of DATAC CASSETTE INTERFACE FIRMWARE ©.

The Program is written so that it must reside in pages 2 and 3, however note that all absolute addresses are underlined, so it may be relocated by simply offsetting the absolute addresses. Note that the program is divided into four parts: WRITE, READ, TEST WRITE, and TEST READ. We will discuss each section in detail.

CASSETTE WRITE

```
0200  A2  FF  9A  20  94  02  20  CA
0208  02  20  9F  02  A5  04  20  55
0210  02  D8  A0  00  84  05  A5  00
0218  20  55  02  A5  01  20  55  02
0220  38  A5  02  E5  00  AA  A5  03
0228  E5  01  D0  1F  8A  F0  56  20
0230  55  02  B1  00  20  AA  02  20
0238  55  02  E6  00  D0  02  E6  01
0240  CA  D0  EF  A5  05  20  55  02
0248  4C  09  02  A2  FF  8A  4C  2F
0250  02                  48  48  A9
0258  05  85  07  68  20  70  02  20
0260  B2  02  A9  05  85  07  68  48
0268  20  71  02  20  B2  02  68  60
0270  2A  C6  07  F0  0B  2A  90  04
0278  29  FE  B0  F4  09  01  90  F0
0280  60  18  F0  FD
0288
0290                  A9  FF  8D  00
0298  14  A9  1D  8D  00  14  60  A2
02A0  FF  A9  BA  20  B2  02  CA  D0
02A8  FA  60  18  48  65  05  85  05
02B0  68  60  48  AD  00  14  29  02
02B8  F0  F9  68  8D  01  14  60  EA
02C0  EA  EA  EA  EA  EA  EA  18
02C8  F0  FD
02D0
```

CASSETTE READ

```
02D8  00  00  00  00  00  00  00  00
02E0  A2  FF  9A  20  94  02  A2  07
02E8  20  2C  03  C9  BA  D0  F7  CA
02F0  D0  F6  20  2C  03  C9  BA  F0
02F8  F9  20  39  03  C5  04  D0  E8
0300  20  36  03  85  00  20  36  03
0308  85  01  20  36  03  85  06  F0
0310  3A  A0  00  84  05  20  36  03
0318  20  AA  02  91  00  C8  C4  06
0320  D0  F3  20  36  03  C5  05  D0
0328  ID  4C  E6  02  AD  00  14  4A
0330  90  FA  AD  01  14  60  20  2C
0338  03  29  55  85  07  20  2C  03
0340  29  55  0A  05  07  60  4C  46
0348  03  EA  EA  EA  EA  4C  BF  02
0350  20  94  02  20  94  02  20  70
0358  03  20  90  03  4C  56  03  EA
0360  20  94  02  20  2C  03  C9  BA
0368  D0  F9  20  70  03  4C  60  03
0370  A9  01  8D  01  6E  A0  FF  AD
0378  00  6E  49  01  8D  00  6E  A2
0380  FF  CA  D0  FD  88  D0  F0  60
0388
0390  A0  FF  A2  FF  CA  D0  FD  88
0398  D0  F8  60
```

ALIGNMENT

We will begin our venture by making the necessary adjustments, that is the recorder volume adjustment, and if you installed a trimmer pot in place of R46 and 47 we will indicate how to adjust it.

First make sure you have your cassette recorder, a good blank cassette tape, cables connected from recorder to the Datac 1000 card, a VOM or oscilloscope and a speaker connected to PAØ of the 6530 (the "music" setup).

Load in the parts of the program that are boxed in, then check to make sure it is loaded correctly. (The boxed in portions of the program are the parts necessary for testing.)

We will now proceed to make a test tape. If you followed our suggestion and replaced R46 and R47 with a trimmer pot, you can now adjust it.

Begin by setting the reset vector with the starting location for the test write program (0350 per listing, 1953 per EPROM), then reset and run.

The ACIA should be writing a burst of the same character, If you have a scope, you can look at pin #6 of the ACIA, and see the output.

The trimmer that replaces R46 and R47 can now be adjusted. Connect a scope or a meter to the cassette output jack and adjust the trimmer so that you get a peak output of approximately 500 mV on the scope or an average DC reading on your meter of 250 mV. (Note: most recorders provide automatic gain control (AGC) so there is no need to adjust the volume control for recording.)

You may now make a recording of the test pattern. Record the test pattern for about two minutes, then stop the recorder and microprocessor.

We will now proceed to adjust the volume control of the recorder for data reading.

Rewind the cassette. Set the reset vector with the starting location of the test read program, (0360 per listing, 1960 for EPROM), reset, then run. Start your cassette recorder playing the test tape and turn the volume control all the way down. You may now begin turning up the volume control very slowly. As you start to get into the correct range, you will start to get an occasional beep from your "music" speaker. You should attempt to get the most consistent tone possible by increasing or decreasing the volume control.

If you are unable to get "beeps" from your speaker, then load in the following program and start it:

```
0010*  20   03     70    beep  (1970 for EPROM)
0013*  4C   10     00
FFFC   10   00
```

This program should produce a constant tone from your "music speaker"; if it does not, then you may have misloaded the test read and write programs or there is a problem with your board. If a constant tone is produced, then the reason why the test read program did not work with your test tape is that your cassette recorder is probably inverting the output (about half of the recorders we tested are in this category). To fix this problem, cut the trace from U13 pin 13 to pin 2 of the ACIA and reconnect that trace to U13 pin 10. This fix just switches the input to the ACIA from the Q output of the flip flop to the $\bar{Q}$ output.

Now repeat the volume adjustment procedure.

WRITING A PROGRAM

Assuming you have a program stored in memory, we will now write it on to tape. Begin by writing beginning and ending addresses of the program you wish to store on tape in the following location:

```
Address 0000  ll (LSB)
        0001  pp (MSB)  = starting address

        0002  ll (LSB)
        0003  pp (MSB)  = ending address
```

Then select an identification number (any number from 00 to FF). Note: you may not use BA. Enter that.

```
Address  0004   ID #
```

Location 0005, 0006, and 0007 are reserved for temp storage and may not be used. Enter starting address of the cassette write program (0200 per listing or 1800 for EPROM) into the reset vector, turn your cassette recorder into into its record mode (wait to make sure the tape leader passes), touch the reset then run pads and you are off.

The address and data lights may flicker, and when the "write" is completed either all the address LEDs will be on or the address LEDs will be on in the pattern 7EF7.

READING A PROGRAM

Now that you have a program recorded on cassette tape, and you have adjusted the recorder volume control, you can read it back into your computer's memory.

To do this, you must enter the ID number of the program into location 0004, set the reset vector to the starting address of the read program (per listing 02E0, 18E0 for EPROM) reset, then run, set your recorder to play.

Some LEDs may flicker. If the read was successful the address LEDs will all be lit or display 7EF7. If there was a checksum error the LEDs will display 194F.

If the read resulted in a checksum error or an incorrect display, it is most likely that the problem is due to a misadjusted volume control, which may be corrected by moving the volume control either up or down and repeating the read procedure until the read is successful.

WHICH CASSETTE TAPE?

Which type of cassette tape is best? We have found using the higher quality audio cassette tape the most reliable, although the cheap (5 for $1.00) tapes will (most of the time) work, it may be difficult to adjust the volume control.

A WORD OR TWO ON OUR CASSETTE EPROM

We have available a 2708 EPROM with our cassette interface program "burned" into it. However, we have added some extra "goodies" to the software. The EPROM contains some useful subroutines, such as a random number generator, a beep subroutine, and for those of you who have terminals we have written a complete cassette monitor program (in the same EPROM) which permits you to communicate through your terminal in plain language and direct the cassette programs. The monitor enables you to automatically load in more than one file. It also turns on and off one of the parallel I/O lines, thereby affording you motor control for your cassette recorder.

The EPROM can be used with or without a terminal, and is available from Datac for $40.00 post paid.

FEEDBACK

We would appreciate hearing your likes and dislikes about our system and also reports on the performance of cassette recorders and tapes that you are using.

The DATAC Connection

72 PIN CARD EDGE CONNECTOR

All of the signals required for system expansion are provided on the 72 pin card edge connector. This connector has 36 pins on either side of the board on 0.156 inch centers. The pin numbering for this connector has pin 1 on the solder side to the left and numbers across to 36. On the component side the connector starts out with the letter A on the left side and letters follow to R on the right. Rev 1 boards have the numbers 1 and 72 printed in copper on the top side of the board. Don't let this fool you. The pin numbering system and signal names are shown in figure 1.

COMPONENT OR TOP SIDE OF PC BOARD

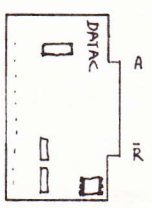
		I-INPUT O-OUTPUT
A	WE	U4-5 O
B	R/W	U4-13 O
C	OD	U4-3 O
D	Ø2	U4-8 O
E	SO	U6-38 I
F	AB15	O
H	AB14	O
J	AB13	O
K	AB12	O
L	AB8	O
M	AB9	O
N	AB10	O
P	AB11	O
R	FIELD Ø	U9-15 O
S	FIELD 2	U9-14 O
T	· 4	U9-13 O
U	· 6	U9-12 O
V	· 8	U9-11 O
W	FIELD A	U9-10 O
X	FIELD C	U9-9 O
Y	GROUND BUS	
Z	+5V BUS	
Ā	ØI OUT	U6-3 O
B̄	IRQ	U6-4 I
C̄	RST	U6-40 I
D̄	+12V BUS	
Ē	UNASSIGNED (RESERVED)	
F̄		
H̄		
J̄		
K̄		
L̄		
M̄		
N̄	-12V BUS	
P̄	+7.5-10V RAW	
R̄	DC POWER BUS	

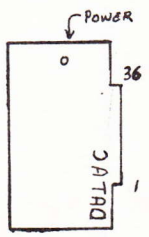
FIGURE 1A DATAC 1000 SYSTEM BUS TOPSIDE

SOLDER OR BOTTOM SIDE OF PC BOARD

		I-INPUT O-OUTPUT
36	+7.5 to 10V RAW DC POWER BUS	
35		
34	HOLD U38-4	I
33	UNASSIGNED (RESERVED)	
32		
31		
30		
29		
28		
27	+12V BUS	
26	-5V BUS	
25	READY U37-3	O
24	SYSBUSIN U12-6	O
23	+5V BUS	
22	GROUND BUS	
21		
20	SYSBUSOUT U12-3	O
19	FIELD E U9-7	O
18	NMI U6-6	I
17	D7 U2-12	I/O
16	D6 U2-4	I/O
15	D5 U2-15	I/O
14	D4 U2-1	I/O
13	D3 U1-12	I/O
12	D2 U1-4	I/O
11	D1 U1-15	I/O
10	DØ U1-1	I/O
9	GROUND	
8	AB7	O
7	AB6	O
6	AB5	O
5	AB4	O
4	AB3	O
3	AB2	O
2	AB1	O
1	ABØ	O

FIGURE 1B DATAC 1000 SYSTEM BUS BOTTOM SIDE





40 PIN RIBBON CONNECTOR

The 40 pin ribbon connector is provided to interface the DATAC 1000 I/O lines to the rest of the world. It is designed to take a standard header with 0.025 square pins on 0.1 inch spacing in two rows (spaced 0.1 inch) of 20. The pins are numbered starting in the lower left hand corner of the board from left to right alternating between the rows. The pin numbering is shown in Figure 2 along with the signal assignments.

Please note that this numbering system is not the same as the one used in earlier issues of the newsletter.

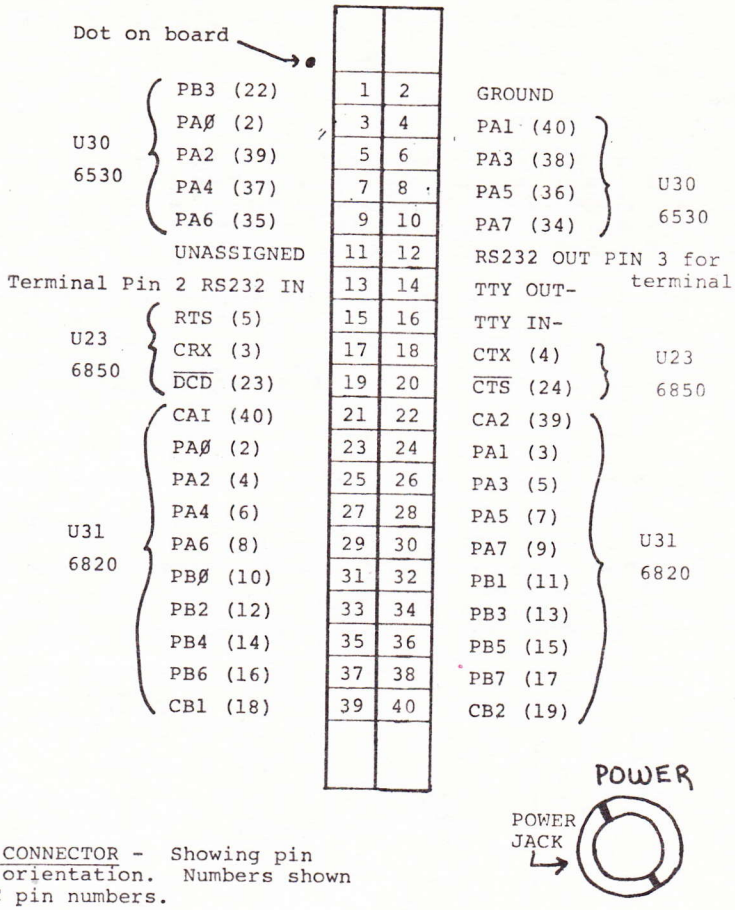


FIGURE 2 RIBBON CABLE CONNECTOR - Showing pin numbering, signals, and orientation. Numbers shown in parenthesis () are IC pin numbers.

Overspeed Clocks

by John Prenis

The original RC values for the Datac board were selected to give it a clock rate of 750 Khtz. A comparison of 1 Mhtz. crystal and RC clocked boards, however, made it appear that the RC equipped boards were faster. Here is a program you can use to tell if your board is speeding.

1 Minute Time Delay

```
Special Registers
0001   reserve for outer loop counter
0002      "     "  middle  "     "
0003      "     "  inner   "     "
FFFC*  10  00

0010   A9  78      2           A=78
0012   85  01      3           (Z01)=A
0014   A9  C8      2   OTR     A=C8
0016   85  02      3           (Z02)=A
0018   A9  FA      2   MID     A=FA
001A   85  03      3           (Z03)=A
001C   EA          2   INR     J+1
001D   C6  03      5           (Z03)-1
001F   D0  FB      3           if≠0,j:INR
0021   C6  02      5           (Z02)-1
0023   D0  F3      3           if≠0,j:MID
0025   C6  01      5           (Z01)-1
0027   D0  EB      3           if≠0,j:OTR
0029   4C  29  00              j rope
```

This program consists of three nested loops and the whole thing takes a total of 59988004 cycles. On a board with a 1 Mhtz. crystal, it takes 59.9 seconds. On a board with an RC clock it should take 80 seconds. When I ran it on my board, it took only 45 seconds, meaning that my board was running at 1.25 Mhtz. Most 6502's can run at 2 Mhtz. or more, so this would not be a problem except that the 6530 and 20 will not work reliably much past 1 Mhtz. Several other boards have been measured and found to be overspeed, so this may explain some of the early problems encountered with these I/O chips. Removing the 18K clock resistor R8 from my board, and trying other values, I found that with a 33K resistor the program ran in 57 seconds (1.05 Mhtz.), with 39K 92 seconds (.65 Mhtz.), with 43K 100 seconds (.6 Mhtz.). Component tolerances probably cause considerable variation. Of course, if you want to be sure, you can always install a crystal.

Personal Finances

Edwin R. Morris

Foreword

If, in your financial accounting, you make a practice of dividing your assets into several accounts, e.g., Tithes, Taxes, Toggery, etc., this program can help in the chore. It operates in the decimal mode, there is no need to translate financial figures to hexadecimal.

The program operates either as an accounting machine (a/c) or an adding machine (adm). As an a/c, it will accept input data for old balance, income, and expense and calculate the new balance. It can be directed to accumulate the new balances in a grand total, or it can be directed to omit adding the balance of any individual account to the grand total.

As an adm, it will accept positive or negative numbers as input, without requiring that negative numbers be complemented.

Register 0000 is loaded by the operator, and its contents set the mode in which the program operates.

Reserved Registers:

0000 Set by operator for program direction:
=00 for accounting operation; =01 for skip grand total; =AA for adding machine use.

The following data registers are in groups of 4 to accomodate 4 pairs of digits. In each case, the lowest numbered register contains the least significant pair of digits, the next higher numbered register the next higher pair of digits, etc. The maximum capacity of the machine is $999,999.99. The decimal point is always assumed to be between the second and third lowest digits.

0001
0002 Registers for old balance for accounts.
0003
0004

0005
0006 Registers for grand total.
0007
0008

0011
0012 Income for accounts, or
0013 positive figures for input to adding machine.
0014

0015
0016 Adding machine total.
0017
0018

0021
0022 Expense for accounts, or
0023 negative figures for input to adding machine.
0024

0031
0032 New balance for accounts.
0033
0034

Program:

(This part of the program sets the data registers to zero.)

```
FFFC*  36 00
oo36*  F8          fd=1          Decimal mode
  37*  A0 00       Y=00
  39   A2 00       X=00          Reset counter
  3B   E8     re1  X+1           Increment counter
  3C   94 04       (Z04+X)=Y     Reset grand total
  3E   E0 04       f1: X-04      Test counter
  40   D0 F9       if/0,j re1    When count is 4 go on
  42   A2 00       X=00          Reset counter
  44   E8     re4  X+1           Increment counter
  46   94 14       (Z14+X)=Y     Reset adm total
  48   E0 04       f1:X-04       Test counter
  4A   D0 F8       if/0,j re4    When count is 4 go on
  4C   A2 FF  nxt  X=FF
  4E   9A EA       ps=X          Stack pointer to FF
  50   A0 00       Y=00
  52   A2 00       X=00          Reset counter
  54   E8     re2  X+1           Increment counter
  56   94 00       (Z00+X)=Y     Reset old balance
  58   94 10       (Z10+X)=Y     Reset income
  5A   94 20       (Z20+X)=Y     Reset expense
  5C   94 30       (Z30+X)=Y     Reset new balance
  5E   E0 04       f1:X-04       Test counter
  60   D0 F2       if/o,j re2    When count is 4 go on
  62   A9 6A       A=6A
  64   8D FC FF    (FFFC)=A      To restart at 006A
0067   4C 67 00    jump rope     Awaits operator's actio
```

Operator touches Halt, and reads grand total if it is desired. If operator wants to use the program for an adm, he sets the value AA in register 0000. He then loads a number (if positive, in registers 0011, 0012, 0013, & 0014; if negative, in registers 0021, 0022, 0023, & 0024), and touches Reset, Run. He again touches Halt, and continues t load numbers as above. When there are no more numbers to b added, he extracts the total from registers 0015, 0016, 0017, & 0018. He then loads 00 in register 0000, 42 in register FFFC, and touches Reset, Run. This resets the adm total and returns to this same spot in the program.

If operator wants the program for an a/c, instead of an adm, he loads the account data as explained in the Foreword, then touches Reset, Run.

(The next part of the program makes account calculations)

```
006A   A0 18       Y=18          Loads operand fc=0
  6C   A9 00       A=00
  6E   85 C7       (ZC7)=A       Changes subroutine
  70   A9 75       A=75
  72   85 CA       (ZCA)=A       "       "
  74   A9 10       A=10
  76   85 CB       (ZCB)=A       "       "
  78   A9 30       A=30
  7A   85 D5 EA    (ZD5)=A       "       "
  7D   20 C0 00    j:subroutine  Adds income to account
  80   A0 38       Y=38          Loads operand fc=1
  82   A9 30       A=30
```

```
 84   85 C7          (ZC7)=A        Changes subroutine
 86   A9 F5          A=F5
 88   85 CA          (ZCA)=A          "         "
 8A   A9 20          A=20
 8C   85 CB          (ZCB)=A          "         "
 8E   20 C0 00       j:subroutine   Subtracts expense fr acct
 91   A9 AA EA       A=AA
 94   C5 00      fl: A-(Z00)        Test for adm
 96   F0 1E          if 0, j:add    To adding machine
 98   A9 A0          A=A0
 9A   8D FC FF       (FFFC)=A       To restart at 00A0
009D  4C 9D 00       jump rope      Awaits operator's action
```

(If the program is in the adm mode, the above jump rope is skipped. Next pause is at program step 0067.)

If program is in a/c mode, operator touches Halt; extracts new balance for account from registers 0031, 0032, 0033, & 0034. If no error is detected, and (0000)=00, operator touches Reset, Run. The program then adds new balance to grand total, and returns operator to program step 0067.

If an error is detected, and the new balance(containing an error) should not be added to the grand total, load 01 into register 0000 and touch Reset, Run. This will return operator to program step 0067, where correct data for the same account can be entered again.

(If in a/c mode, the next part of program adds account balance to grand total. If in adm mode, it adds the input to adm total.)

```
00A0   A5 00          A=(Z00)
 A2    D0 0F          if/0, j:skp    To avoid grand total
 A4    A9 04          A=04
 A6    85 CB          (ZCB)=A        Changes subroutine
 A8    85 D5          (ZD5)=A          "         "
 AA    A0 18          Y=18           Loads operand fc=0
 AC    A9 75          A=75
 AE    85 CA          (ZCA)=A        Changes subroutine
 B0    20 C0 00       j:subroutine   Adds acct to gnd total
 B3    4C 4C 00   skp j:nxt
 B6    A9 14      add A=14
00B8   4C A6 00
```

SUBROUTINE:

```
00C0   A2 00          X=00           resets counter
 C2    E8 EA      re3 X+1            Increments counter
 C4    84 C8          (ZC8)=Y        Sets fc for step 00C8
 C6    B5 B5          A=(Zll+X)      Set by program
 C8    B5 EA                         See steps 00C4 & 00C5
 CA    EA EA          A:fc:(Zll+X)   Set by program
 CC    B0 02          if fc=1, j:car
 CE    A0 18          Y=18
 D0    90 02      car if fc=0, j:ncr
 D2    A0 38          Y=38
 D4    95 95      ncr (Zll+X)=A      Set by program
 D6    E0 04          fl: X-04       Test counter
 D8    D0 E8          if/0, j:re3    When count is 4 go on
00DA   60             j:ret from subroutine.
```

Good Programming Practices

by John Prenis
from the Datac 1000 Tutorial Manual

This time we'll continue our discussion of programming structures with a look at loops.

Another structure of great importance is the loop with a conditional exit, often called a "do-while" or "do-until" loop. Here is a simple loop in TLC:

```
LOOP
   DO:THIS:THING
   EXIT IF N=0
POOL
```

Each time the routine DO:THIS:THING is executed, N is checked. If it is not zero, the program goes to the statement following LOOP. As soon as the routine causes N to be zero, the program goes to the next statement after POOL ("LOOP" spelled backwards). Once again, the use of indentation and of special words helps to make the program clearer. Here is a similar loop in 6502 machine language:

```
1030   20  ( )( )   A   call DO:THIS:THING
1033   A5  09           load accumulator with N
1035   D0  F9           if N≠0, go to A
1037   ....             program continues
```

The last "structure" we'll consider is a simple sequence of instructions with no loops or branches. We'll call this a linear structure.

Actual experience has shown that these structures alone are sufficient for all applications. From the programmer's point of view there are several advantages to using a limited set of control structures. A small set of structures is quickly learned. Because he is familiar with their workings, the programmer can use them with confidence and a minimum of errors. The programmer is also relieved of the constant need to verify the correctness of non-standard control structures.

There are some disadvantages. The structured solution is not always obvious. However it is usually worth the extra effort to find one. Structured programs take up more memory and seldom run as fast as "optimized" programs. These disadvantages are outweighed by the ease with which structured programs are debugged and maintained.

The full benefits of structured programming are obtained chiefly throuth the use of a high level language designed with structured programming in mind. However the principles are also useful in machine language programming. We have already seen how the necessary structures can be carried out in machine language. We cannot do without the go-to on the machine language level, but by imposing restrictions on its use, we can keep it from getting us into trouble.

1. Go-to's should be used to jump forward only.
2. Go-to's should be used only within a module, never between modules.
3. Whenever possible, go-to's should go to the exit of the module in which they are used.

There are two main exceptions to these rules. When a program is written in the form of a big loop that is continuously executed as long as the machine is turned on, it is all right to use a backward go-to to close the loop. The other exception is the use of a go-to to "stretch" a branch that must go more than 127 bytes forward or 128 bytes backward.

These guidelines are not intended as hard and fast rules. You will probably run into situations where a judicious use of the go-to can make your code more straight forward. The important thing is to be aware of what you are doing.

Although top-down design, modular programming, and structured programming are independent techniques, you may have noticed that they work together very nicely. People who get tired of saying "top-down-structured-modular-programming" all in one breath tend to lump them all together under the name structured programming. I look forward to the day when we won't need any special names. We'll just think of them as good everyday programming practices.

The ideas just discussed don't begin to exhaust the area of good programming practices. Here are some more:

The very first step in writing a program is to define in as much detail as possible exactly what the program should do. Include samples of input and output if possible. Details of how the program will do its job should be left til later - the first thing to settle is how the program will look to the outside world.

Think before you program. Consider alternatives. Professional programmers spend far more time thinking than they do in writing code.

Make extensive use of subroutines as procedures. Your program will be reduced to little more than a series of subroutine calls. Give each call an English name and you have a program that is brief, easy to read, and easy to understand. The details are hidden away on a lower level, as top-down design requires. The logical structure of the program becomes clearer. The program also becomes easier to debug. By using dummy routines, you can test the main program even before the subroutines are written.

Avoid tricks. It's tempting to show how clever you are by taking advantage of quirks in the processor's instruction set or by writing code that modifies itself. Don't do it. Always use the most straightforward method. Tricks are difficult to explain, hard to debug, almost impossible to document.

Avoid byte squeezing. A common amusement of programmers is to see who can write the shortest or the fastest program to do such-and-such. This is OK for fun, but keep it out of your serious work. The only time it's justified is when your program is just a few bytes too long to fit into memory, or when a real-time program is just a little bit too slow. Consider buying more memory or a faster processor.

Save results in memory, not the accumulator. When carrying results from one routine to the next in the accumulator, you must be very careful about what each routine expects to find in the accumulator and what it leaves. Changes become difficult. When you store intermediate results in memory, you know where they are and what they are. You have greater freedom because they are available to all routines at any time.

Don't be stingy with working registers. Give each variable its own. Trying to save two variables in the place at different times makes it difficult to remember which is where when, and you may get B when you really wanted A.

Build debugging techniques into your programs. One good idea is to store intermediate results in places where they can be examined later. Another is to leave no-ops in your programs where breakpoints can be inserted. When the program reaches one of these, it jumps to a routine that tells you just what the program has been doing lately. A good place to put them is at the end of each module. When you are done checking a module, you turn the breakpoint instruction back into a no-op. The extra instructions can be removed later if necessary.

Check your program by hand before running it. You do this by sitting down with the program and following its instructions exactly, step by step, as the computer would. It's embarrassing how often this roots out a mistake that would have caused the program to fail. Single stepping through your program before running it is a good idea too.

Debug programs from the top down. You want to be sure that the main program is working correctly before checking the subroutines.

Consider all the possible conditions your routines will face. What happens if the data is bad, or out of range, or missing? What happens if the user hits the wrong key or when some smart alec types a negative number when asked for a positive one? Failure to keep such things in mind can result in a program that seems to run all right, but which later blows up unexpectedly when presented with something the programmer didn't forsee.

Don't patch. Sometimes you need to insert a few extra lines into a program. The simple way is to use a go-to to jump to some part of memory where there is room, put the extra instructions there and then use another go-to to jump back into the program. After this is done a few times, the program becomes an incomprehensible jumble. If you are using procedures properly, there is no need for this. Just rewrite the procedure and relocate it to a section of memory where there is room for it. (Don't forget to change the call from the main program.)

Start over if you have to. Salvaging a poor program can cost far more effort than simply rewriting it. Besides, you'll do a much better job the second time around.

Document your program when you have it working It is a rare program that will not have to be changed or updated someday, and you will certainly not remember how today's program works a month from now. Describe briefly what the program does. Use comments to tell you what the program is doing (not what the processor is doing). Make the logical structure of the program clear by skipping lines, indenting , or circling blocks of code with a pencil. Make a list of what variables are stored in what registers. Tell what the program expects as input, what it delivers as output, and what it does in case of error. You'll find the time well spent.

BIBLIOGRAPHY

Ledgard, Henry

Programming Proverbs

Hayden Book Co. 1975

Karp, Tony

"TLC- A New Systems Language"

Proceedings, uPIEE-77 Workshop on Bench Programming of Microprocessors

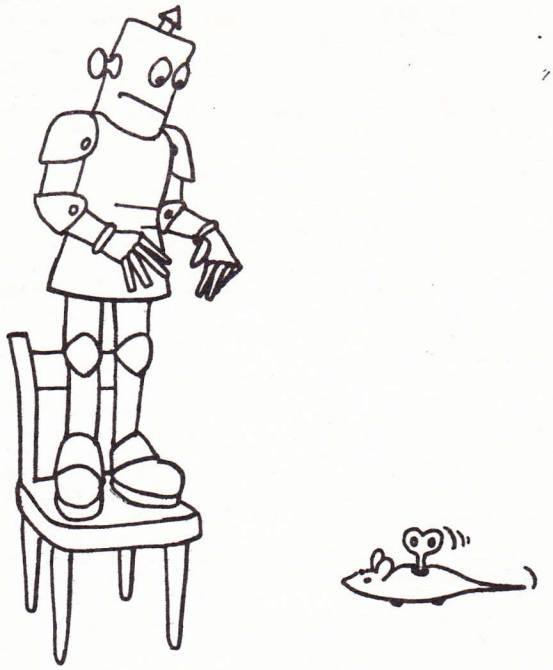
Jones, Bill

"Structured Programming"
Kilobaud, vol. 1, no. 5 May 1977

Chamork, Glen

"Structured BASIC is Better!"

Kilobaud, vol. 1, no. 1 January 1977



DATAC ENGINEERING
P.O. BOX 406
SOUTHAMPTON, PA. 18966

FIRST CLASS