TRSTimes

Volume 7. No. 5 - Sep/Oct 1994 - $4.00

DR. PATCH

UTILITY FOR TRS-80 MODEL 4 AND LS-DOS 6.3.1

A *'MUST HAVE'* FOR ALL LS-DOS 6.3.1 OWNERS.

DR. PATCH MODIFIES LS-DOS 6.3.1 TO DO THINGS THAT WERE NEVER BEFORE POSSIBLE.

COMPLETELY SELF-CONTAINED - MENU-DRIVEN FOR MAXIMUM USER CONVENIENCE.

FAST & SAFE - EACH MODIFICATION IS EASILY REVERSED TO NORMAL DOS OPERATION.

DISABLE PASSWORD CHECK IN FORMAT/CMD
FORMAT DOUBLE-SIDED AS DEFAULT
FORMAT 80 TRACKS AS DEFAULT
DISABLE VERIFY AFTER FORMAT
CHANGE 'DIR' TO 'D'
CHANGE 'CAT" TO 'C'
DIR/CAT WITH (I) PARAMETER AS DEFAULT
DIR/CAT WITH (S,I) PARAMETERS AS DEFAULT
CHANGE 'REMOVE' TO 'DEL'
CHANGE 'RENAME' TO 'REN'
CHANGE 'MEMORY' TO 'MEM'
CHANGE 'DEVICE' TO 'DEV'
DISABLE THE BOOT 'DATE' PROMPT
DISABLE THE BOOT 'TIME' PROMPT
DISABLE FILE PASSWORD PROTECTION
ENABLE EXTENDED ERROR MESSAGES

DISABLE PASSWORD CHECK IN BACKUP/CMD
BACKUP WITH (I) PARAMETER AS DEFAULT
BACKUP WITH VERIFY DISABLED
DISABLE BACKUP 'LIMIT" PROTECTION
DISABLE PASSWORD CHECK IN PURGE
PURGE WITH (I) PARAMETER AS DEFAULT
PURGE WITH (S,I) PARAMETERS AS DEFAULT
PURGE WITH (Q=N) PARAMETER AS DEFAULT
IMPLEMENT THE DOS 'KILL' COMMAND
CHANGE DOS PROMPT TO CUSTOM PROMPT
TURN 'AUTO BREAK DISABLE' OFF
TURN 'SYSGEN' MESSAGE OFF
BOOT WITH NON-BLINKING CURSOR
BOOT WITH CUSTOM CURSOR
BOOT WITH CLOCK ON
BOOT WITH FAST KEY-REPEAT

DR. PATCH IS THE ONLY PROGRAM OF ITS TYPE EVER WRITTEN FOR THE TRS-80 MODEL 4 AND LS-DOS 6.3.1.

DISTRIBUTED EXCLUSIVELY BY TRSTIMES MAGAZINE ON A STANDARD LS-DOS 6.3.1 DATA DISKETTE, ALONG WITH WRITTEN DOCUMENTATION.

DR. PATCH $14.95

NO SHIPPING & HANDLING TO U.S & CANADA. ELSEWHERE PLEASE ADD $4.00
(U.S CURRENCY ONLY, PLEASE)

TRSTimes magazine - dept. DP
5721 Topanga Canyon Blvd. #4
Woodland Hills, CA 91367

DON'T LET YOUR LS-DOS 6.3.1 BE WITHOUT IT!

TRSTimes magazine

Volume 7. No. 5 - Sep/Oct 1994 - $4.00

PUBLISHER-EDITOR
Lance Wolstrup
CONTRIBUTING EDITORS
Roy T. Beck
Dr. Allen Jacobs
TECHNICAL ASSISTANCE
San Gabriel Tandy Users Group
Valley TRS-80 Users Group
Valley Hackers' TRS-80 Users Group

TRSTimes is published bi-monthly by TRSTimes Publications. 5721 Topanga Canyon Blvd., Suite 4. Woodland Hills, CA 91367. U.S.A. (818) 716-7154.

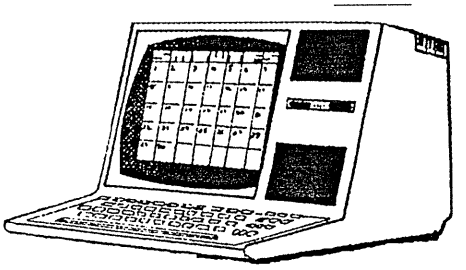
Publication months are January, March, May, July, September and November.

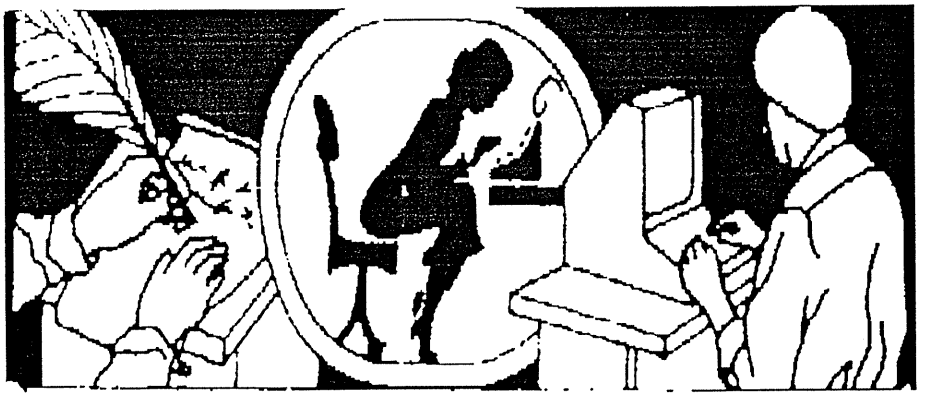
1994 subscription rates (6 issues):
UNITED STATES & CANADA:
$20.00 (U.S. currency)

EUROPE, CENTRAL & SOUTH AMERICA:
$24.00 for surface mail or $31.00 for air mail. (U.S. currency only)

ASIA, AUSTRALIA & NEW ZEALAND:
$26.00 for surface mail or $34.00 for air mail. (U.S. currency only)

Article submissions from our readers are welcomed and encouraged. Anything pertaining to the TRS-80 will be evaluated for possible publication. Please send hardcopy and, if at all possible a disk with the material saved in ASCII format. Any disk format is acceptable, but please note on label which format is used.





TRSPHONE

Model 4 - editor/assembler (EDAS)

by Lance Wolstrup

This month, I thought I'd put together something useful - a phonelist utility for Model 4. It is certainly something that I can use myself, as I am undoubtedly the most unorganized guy west of Mississippi. My desk is always cluttered with scraps of paper containing phone numbers of people that I need to call back for one reason or another. Needless to say, I lose most of them - that is, until I began using my Mod 4 to store the numbers.

I know that I could have used a big database, such as Profile or PFS/file, to do the job, but that was too clunky. I really didn't want to load a program and then go through a series of menus to find the information; rather, I wanted to be able to type a name from the DOS prompt - having my program display that person's phone number. Nothing fancy, just quick and dirty — with the emphasis on quick.

I got out my favorite editor/assembler, EDAS, and sat down in front of my Model 4P. The result of the programming session is PHONE/CMD, a short machine language program that searches an ASCII datafile for the name specified at the command line and, if found, displays the full name and phone number.

PHONE/CMD is designed to read a plain ASCII datafile containing your phone list. I used TED, but you can use any text editor or wordprocessor, as long as it is capable of saving the text in ASCII. The list must be written uniformly with each line having a person's name, a space, a phone number, and a carriage return indicating the end of the line. For example:

Paula Jones (209) 555-1234 <cr>
Kathy Ferguson (505) 555-4321 <cr>
George Putnam (213) 555-3412 <cr>
Larry Nicholls (318) 555-2143 <cr>
TRSTimes magazine (818) 716-7154<cr>
etc.

After the names and phonenumbers have been entered, save the file as PHONE/DAT. You are now ready to use PHONE/CMD. From the DOS command line, simply type:

PHONE *name* <cr>

For example, typing:

PHONE TRSTimes <cr>

would display:

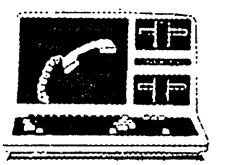
TRSTimes magazine (818) 716-7154

Since PHONE/CMD searches PHONE/DAT for whatever name you specify following PHONE, you need not type the entire name - you only need to type enough characters to make the request unique; otherwise, the first match will be displayed. Also, PHONE/CMD's search function is not case-sensitive; you may type the name request in upper-case, lowercase, or any combination thereof.

The concept of the program is simple. You type the name of the program (PHONE), a space, and then all or part of the name you wish to display. PHONE/CMD reads the contents of the PHONE/DAT datafile into a buffer and then proceeds to compare the name parameter to the beginning of each record until there's either a match or the end of the datafile is detected. If a match is found, the record is displayed on the screen and the program returns to DOS. If no match is found, PHONE/CMD simply returns to the DOS prompt.

This is one segment of the program that I would like to see improved by one or more of the readers. If no parameter follows the program name, I think it would be a good idea to display the entire list, one page at a time.

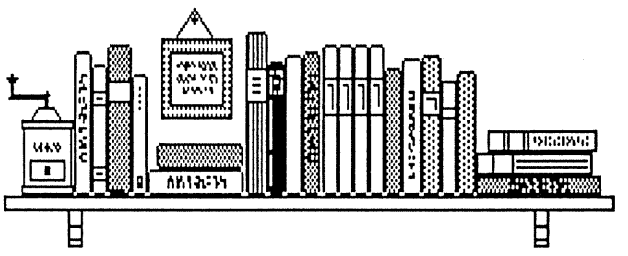
Consider it our fall programming challenge, so get out the editor/assembler of your choice, type in the program, and then modify it as described. If anyone takes the challenge, we will publish the changes in the next issue.



PHONE/ASM

```
;phone/asm
;copyright 1994 by Lance Wolstrup
;for TRS-80 Model 4
;
;
        ORG     3000H
START LD        A,(HL)          ;get param
;
;the next two lines exits to DOS if no name
;was typed. These two lines should be eliminated
;if the proposed modification of displaying the
;entire list is to be implemented.
;
        CP      13              ;none there?
        RET     Z               ;exit if not
;
;save pointer to name and convert the name
;to uppercase
;
        PUSH    HL              ;save params
STRT1 CP        97              ;start of lowercase
        JR      C,STRT2         ;jump if smaller
        CP      123             ;end of lowercase
        JR      NC,STRT2        ;jump if = or >
        RES     5,(HL)          ;make uppercase
STRT2 INC       HL              ;next chr
        LD      A,(HL)          ;xfer to A
        CP      13              ;is it terminator
        JR      NZ,STRT1        ;no, so repeat
;
;restore the name pointer and store address
;in the NAME buffer.
;
STRT3 POP       HL              ;restore params
        LD      (NAME),HL       ;and save it
;
;setup FCB
;
        LD      HL,DATNAM       ;point to datafile
                                ;name
        LD      DE,FCB          ;point to fcb
        LD      A,78            ;@fspec
        RST     40
;
;now attempt to open PHONE/DAT
;
        LD      HL,IOBUF        ;point to i/o buffer
        LD      A,59            ;@open
        LD      B,0             ;reclen 256
        RST     40
;
;determine if OPEN succeeded
;
        JR      Z,OPOK          ;jump if file opened
;
;OPEN failed. Display error message and return
;to DOS.
;
        LD      HL,NOFIL        ;file not found msg
        LD      A,10            ;@dsply
        RST     40
        RET                     ;exit to dos
;
;OPEN succeeded. Get number of 256 byte records
;in PHONE/DAT from FCB. Note that I am only
;reading FCB+12 (the LSB of record counter). I do
;not anticipate using more than 255 256-byte
;records (I figure this to be approximately 1000
;names and phone numbers - ought to be enough!)
;First check if any records exist.
;
OPOK  LD        A,(FCB+12)      ;get number of recs
        OR      A               ;any there?
        JR      NZ,READ         ;yes, so jump
;
;PHONE/DAT has no records, so display message
;and exit to DOS
;
        LD      HL,NREC         ;point to msg
        LD      A,10            ;@dsply
        RST     40
        RET                     ;return to dos
NREC  DB        'No records found in datafile',13
;
;read all records - extend buffer as needed.
;Note that we take some liberties with FCB+3
;and FCB+4 - we keep plugging in the address
;of the new buffer segment.
;
READ  LD        B,A             ;get recs in b
        LD      HL,IOBUF        ;point to i/o buffer
RDLOOP PUSH HL                  ;save it
        LD      DE,FCB          ;point to fcb
        LD      A,67            ;@read
        RST     40
        LD      DE,256          ;extend
        POP     HL              ;i/o buffer
        ADD     HL,DE           ;by 256 bytes
        LD      A,L             ;and copy
        LD      (FCB+3),A       ;new address
        LD      A,H             ;into
        LD      (FCB+4),A       ;fcb
        DJNZ    RDLOOP          ;repeat until done
;
;all records have been read - so close file
;
        LD      DE,FCB          ;point to fcb
        LD      A,60            ;@close
        RST     40
;
;Now we need to determine how many bytes
;in the file - and where the end-of-file marker is.
```

```
;
;
        LD      HL,0            ;set hl to 0
        LD      A,(FCB+12)      ;get number of recs
        DEC     A               ;adjust for eof offset
        OR      A               ;is it now record 0?
        JR      Z,REC0          ;yes, so skip loop
        LD      DE,256          ;inc each rec by 256
        LD      B,A             ;copy number of recs
                                ;to register B
RLOOP ADD       HL,DE           ;add bytes to hl
        DJNZ    RLOOP           ;repeat until done
REC0    LD      A,(FCB+8)       ;end byte offset
        LD      D,0             ;add eof offset
        LD      E,A             ;to
        ADD     HL,DE           ;hl
        LD      DE,IOBUF        ;and then add
        ADD     HL,DE           ;the buffer offset
        LD      (EOF),HL        ;and save eof address
;
        LD      B,0             ;reset loop counter
        LD      HL,(NAME)       ;get param
GET0    LD      A,(HL)          ;get chr
        CP      13              ;is it cr
        JR      Z,GET1          ;yes, so exit loop
        INC     B               ;inc loop counter
        INC     HL              ;next chr
        JR      GET0            ;continue loop
;
GET1    LD      A,B             ;copy loop ctr to a
        LD      (NAMLEN),A      ;and store in buffer
        LD      DE,IOBUF        ;point to i/o buffer
GET2    LD      HL,(NAME)       ;get param
        PUSH    DE              ;save i/o pointer
GET3    LD      A,(DE)          ;get chr
        CP      97              ;is it lowercase
        JR      C,GET4          ;no, jump
        CP      123             ;is it lowercase
        JR      NC,GET4         ;no, so jump
        RES     5,A             ;make uppercase
GET4    LD      C,A             ;copy chr to c
        LD      A,(HL)          ;get chr from param
        CP      C               ;compare them
        JR      NZ,NEXT         ;jump if no match
        INC     HL              ;next param chr
        INC     DE              ;next i/o buffer chr
        DJNZ    GET3            ;continue
;
; match found - display name & phone number
;and then exit
;
        POP     DE              ;restore start
                                ;of record
        EX      DE,HL           ;move it to hl
        LD      A,10            ;@dsply
        RST     40
        RET                     ;return to dos
;
;match not found - but before we can check the
;next record we must take care that we are not
;at the end of the file - if we are then exit to DOS.
;
NEXT    POP     DE              ;restore i/o buffer
                                ;address
        LD      HL,(EOF)        ;get eof address
        SBC     HL,DE           ;compare them
        JR      NC,NLOOP        ;jump if not at end
        RET                     ;at end - so return
                                ;to dos
;
;we are not at end, so loop until we get to the end
;of the record (end of record marker is cr)
;
NLOOP LD        A,(DE)          ;get chr
        CP      13              ;is it cr
        JR      Z,NLOOP1        ;jump if cr
        INC     DE              ;next chr
        JR      NLOOP           ;continue
;
;end of record found, so move pointer to beginning
;of the next record - pick up length of parameter
;and store in in regoster B - then go back and
;try to find a match.
;
NLOOP1 INC      DE      ;move past cr
        LD      A,(NAMLEN)      ;pick up param len
        LD      B,A             ;and copy it to b
        JR      GET2            ;continue with the
                                ;next record
;
;messages and buffers
;
NOFIL           DB      'Unable to find '
DATNAM          DB      'PHONE/DAT',13
NAME            DS      32
NAMLEN          DB      0
;
FCB             DS      32
EOF             DS      2
IOBUF           DS      256
;
                END     START
```



MIXCASE

Model 4, Basic

by G. Michels

This is a program that allows persons with hand and wrist problems to type in all upper case letters (to avoid the trauma of reaching for the shift keys) and convert the text to mixed case prior to printout.

The program accesses your .ASC file saved in ASCII format. Thus all enhancements, graphics, bold letters, special characters etc. should be added to the new mixed case file /ASC. If your word processor uses an extension other than /ASC, modify lines 110, 120, 130 and 140 of the program.

The rules are relatively simple. All letters that follow two or more spaces are automatically capitalized, such as at the beginning of a sentence. All mid-sentence letters that are to be capitalized should be proceeded by a /. If a "/" is used within the text, then precede it by another /. If a letter following two spaces is not to be capitalized, then use a slash before it. Skip a line at the top of the text to avoid picking up the software coding of your word processor. Enhancements (bolds, underlines, etc.) can be added to the new text that has been run through MIXCASE/BAS. Your letterhead can be cut and pasted on at that time as well.

Suggestion: Name original text file1. Mixcase file2. (rtjustify file3 if used).

Once you are accustomed to adding the "/" prior to mid-sentence words or names that are to be capitalized, use of the program becomes a fairly simple routine.

The Basic program listing allows you to change the "/" slash preceding each mid-sentence word that is to be capitalized to any other special character that you find easy to use. Also, remember to change the .ASC extension if it is not appropriate for your word processor.

RTJUST/BAS

This program is a means of right justifying text if that feature is not included with your word processor. If you already have the feature, skip this program since it is kind of a last resort.

ENV3/BAS
ENV4/BAS
ENV5/BAS

Written for a DMP33 printer since setting up labels for every envelope that needs to be mailed gets to be a chore. Hopefully these programs can be modified for your printer if you do not have software that prints envelopes.

ENV3 is for a 3 line address, ENV4 is for a 4 line address -etc.

Before using, modify programs to print your return address where it says "type your name here", "type your street address here","type city,type 2 char state and zip".

Place envelope so the top edge is just below the top edge of the printer head. Align left side just as you would paper. Adjust tension slightly looser than you would for single sheets of paper. You can test these programs on a plain sheet of paper in your printer before actual use to see how they work out for you.

EXAMPLE OF INITIAL TYPING PRIOR TO USE OF MIXCASE/BAS AND RTJUST/BAS

(see above)
THE RULES ARE RELATIVELY SIMPLE. ALL LETTERS THAT FOLLOW TWO OR MORE SPACES ARE AUTOMATICALLY CAPITALIZED - SUCH AS AT THE BEGINNING OF A SENTENCE. ALL MID-SENTENCE LETTERS THAT ARE TO BE CAPITALIZED SHOULD BE PROCEEDED BY A //. IF A "//" IS USED WITHIN THE TEXT, THEN PRECEDE IT BY ANOTHER //. IF A LETTER FOLLOWING TWO SPACES IS NOT TO BE CAPITALIZED, THEN USE A SLASH BEFORE IT. SKIP A LINE AT THE TOP OF THE TEXT TO AVOID PICKING UP THE SOFTWARE CODING OF YOUR WORD PROCESSOR. ENHANCEMENTS (BOLDS, UNDERLINES, PICTURES ETC.) CAN BE ADDED TO THE NEW

TEXT THAT HAS BEEN RUN THROUGH /M/I/X/C/A/S/E./B/A/S. /YOUR LETTERHEAD CAN BE CUT AND PASTED ON AT THAT TIME AS WELL.

MIXCASE/BAS

```
10 'MIXCASE/BAS     copyright G.Michels
15 'THIS PROGRAM CHANGES ALL UPPER-
CASE CHAR.TO MIXED CASE
16 '
20 '32=SP  91=[  47=/  10=LF  13=C/R  65-90=
CAPITAL LETTERS
100 CLS
110 PRINT "WHICH .ASC FILE DO YOU WANT
TO CHANGE TO MIXED CASE LETTERS? "
115 PRINT "/ASC EXTENSION IS ASSUMED
AND NEED NOT BE TYPED IN: "
120 INPUT UP$:UPP$=UP$+"/ASC"
130 PRINT "NAME OUTPUT  /ASC FILE: "
140 INPUT LW$:LOW$=LW$+"/ASC"
145 'KILL LOW$
150 OPEN"R",#2,LOW$,70
160 FIELD #2,70 AS D$
170 OPEN "I",#1, UPP$
180 Y=0:Z=0:N=0
190 A$=INPUT$(1,#1)
200 IF EOF(1) THEN 380
210 IF A$<>CHR$(32) THEN Y=0
220 IF A$=CHR$(91) THEN A$=INPUT$(1,#1):
GOTO 280
230 IF A$=CHR$(10) THEN GOSUB 350:
A$=INPUT$(1,#1):GOSUB 350:GOTO 190
235 'IF A$=CHR$(13) THEN Z=70:GOSUB 350:
GOTO 190
240 IF A$=CHR$(32) THEN Y=Y+1:GOTO 310
250 IF A$=CHR$(47) THEN A$=INPUT$(1,#1):
GOSUB 350:GOTO 190
260 IF ASC(A$)<65 OR ASC(A$)>90 THEN
GOSUB 350:GOTO 190
270 IF EOF (1) THEN 380
280 X=ASC(A$)+32
290 A$=CHR$(X)
300 GOSUB 350:GOTO 190
310 IF Y=1 THEN GOSUB 350:GOTO 190
320 IF Y=2 THEN GOSUB 350:
A$=INPUT$(1,#1):IF A$=CHR$(32) THEN Y=Y+1
330 IF A$<>CHR$(32) THEN Y=0
340 GOSUB 350:GOTO 190
350 IF EOF(1) THEN 380
360 IF Z=70 THEN N=N+1:LSET D$=B$:
PRINT D$;:PUT #2,N:B$="":B$=B$+A$:Z=1:
RETURN
370 Z=Z+1:B$=B$+A$:IF Y=>2 THEN Y=Y+1
375 RETURN
380 N=N+1:B$=B$+A$:LSET D$=B$:PRINT D$;:
PUT #2,N:CLOSE:END
```

ENV3/BAS

```
1 ' ENVELOPE PRINTING WITH RETURN ADDRESS
copyright G.MICHELS
2 '
3 '
10 CLS:
LINE INPUT "ENTER NAME OF ADDRESSEE: ",N$:
PRINT:LINE INPUT"ADDRESS:  ",A$
20 PRINT:LINE INPUT "CITY: ",C$:
INPUT "STATE AND ZIP:",ST$
30 PRINT TAB(30) N$:PRINT:PRINT TAB(30) A$:
PRINT:PRINT TAB(30)C$", "ST$
40 PRINT:INPUT"IS THIS CORRECT? Y/N: ",ANS$
50 IF ANS$ <> "Y" GOTO 10
55 INPUT "PLEASE POSITION ENVELOPE IN
PRINTER. READY? Y/N: ",Y$
57 IF Y$<>"Y" THEN 55
60 LPRINT CHR$(27);CHR$(0);CHR$(2):
70 LPRINT TAB(1)"type your name here":
LPRINT TAB(1)"type your street address here"
80 LPRINT TAB(1) "type city,2 char state zip";CHR$(11)
90 LPRINT TAB(30)"      ":LPRINT TAB(30)"      ":
LPRINT TAB(30)"    "
100 LPRINT TAB(30)"  ":LPRINT TAB(30)"  ":
LPRINT TAB(30) N$:LPRINT TAB(30) A$
110 LPRINT TAB(30) C$ ", " ST$
120 INPUT "PRINT MORE ENVELOPES TO
ADDRESSEE? Y/N: ",ANS$
130 IF ANS$ = "Y" THEN 55
160 END
```

ENV4/BAS

```
1 ' ENVELOPE PRINTING WITH RETURN AD-
DRESS   copyright G.Michels
2 '
3 '
5 CLS:PRINT"4 LINE ADDRESS":PRINT
10 LINE INPUT "ENTER NAME OF ADDRESSEE:
",N$:
PRINT:LINE INPUT"ADDRESS:  ",A$
15 LINE INPUT "2ND LINE ADDRESS: ",A1$
20 PRINT:LINE INPUT "CITY: ",C$:INPUT
"STATE AND ZIP:",ST$
30 PRINT TAB(30) N$:PRINT:PRINT TAB(30) A$:
PRINT:PRINT TAB(30) A1$
35 PRINT:PRINT TAB(30) C$","ST$
40 PRINT:INPUT"IS THIS CORRECT? Y/N:
",ANS$
50 IF ANS$ <> "Y" GOTO 10
55 INPUT "PLEASE POSITION ENVELOPE IN
PRINTER. READY? Y/N: ",Y$
57 IF Y$<>"Y" THEN 55
```

```
60 LPRINT CHR$(27);"C";CHR$(0);CHR$(2):
70 LPRINT TAB(1)"type your name here":
LPRINT TAB(1)"type your street address here"
80 LPRINT TAB(1) "type city, 2char state
zip";CHR$(11)
90 LPRINT TAB(30)"      ":LPRINT TAB(30)"      ":
LPRINT TAB(30)"    "
100 LPRINT TAB(30)"  ":LPRINT TAB(30)"  ":
LPRINT TAB(30) N$:LPRINT TAB(30) A$
110 LPRINT TAB(30)A1$:
LPRINT TAB(30) C$ ", " ST$
120 INPUT "PRINT MORE ENVELOPES TO
ADDRESSEE? Y/N: ",ANS$
130 IF ANS$ = "Y" THEN 55
160 END
```

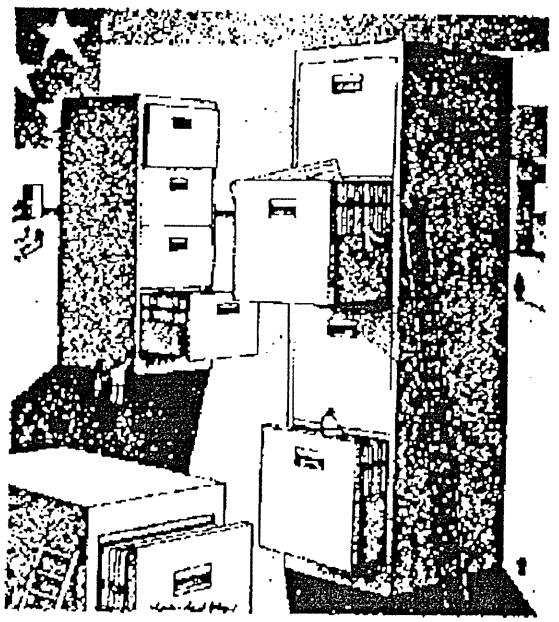
ENV5/BAS

```
1 ' ENVELOPE PRINTING WITH RETURN
ADDRESS  copyright G.Michels
2 '
3 '
5 CLS:PRINT"5 LINE ADDRESS":PRINT
10 LINE INPUT "ENTER NAME OF ADDRESSEE:
",N$:PRINT:LINE INPUT "ADDRESS:  ",A$
15 LINE INPUT "2ND LINE ADDRESS: ",A1$:
LINE INPUT "3RD LINE ADDRESS: ",A2$
20 PRINT:INPUT "CITY: ",C$:
INPUT "STATE AND ZIP:",ST$
30 PRINT TAB(30) N$:PRINT:PRINT TAB(30) A$:
PRINT:PRINT TAB(30) A1$
35 PRINT:PRINT TAB(30) A2$:PRINT:
PRINT TAB(30) C$","ST$
40 PRINT:
INPUT"IS THIS CORRECT? Y/N: ",ANS$
50 IF ANS$ <> "Y" GOTO 10
55 INPUT "PLEASE POSITION ENVELOPE IN
PRINTER. READY? Y/N: ",Y$
57 IF Y$<>"Y" THEN 55
60 LPRINT CHR$(27);"C";CHR$(0);CHR$(2):
70 LPRINT TAB(1)"type your name here":
LPRINT TAB(1)"type your street address here"
80 LPRINT TAB(1) "type city, 2 char state
zip";CHR$(11)
90 LPRINT TAB(30)"      ":LPRINT TAB(30)"      ":
LPRINT TAB(30)"    "
100 LPRINT TAB(30)"  ":LPRINT TAB(30)"  ":
LPRINT TAB(30) N$:LPRINT TAB(30) A$
110 LPRINT TAB(30)A1$:LPRINT TAB(30)A2$:
LPRINT TAB(30) C$ ", " ST$
120 INPUT "PRINT MORE ENVELOPES TO AD-
DRESSEE? Y/N: ",ANS$
130 IF ANS$ = "Y" THEN 55
160 END
```

RTJUST/BAS

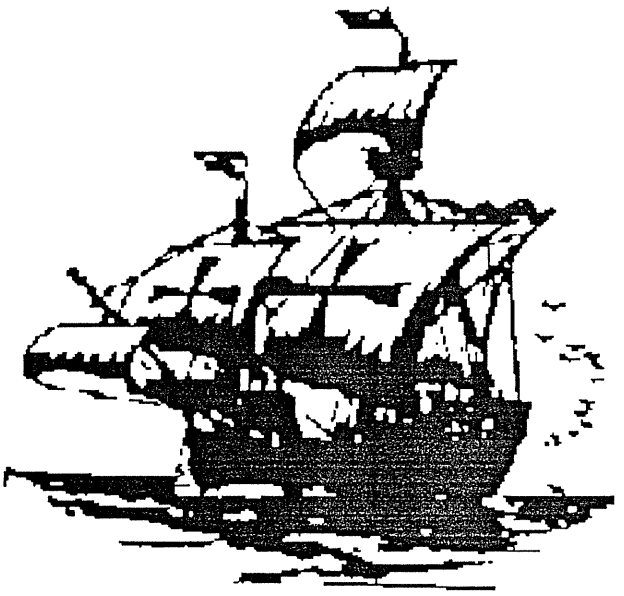
```
10 'RTJUST.BAS   copyright G.Michel
20 '
30 '
100 'A$=ONE CHAR READ
TB=CHAR COUNT PER LINE (CURSOR POS)
110 'LSC(SP)=LINE SPACE COUNT
SP=OCCURANCE OF SPACE IN A LINE
120 'INDENT=0 IF NO INDENT,
CR=CARRIAGE RETURN COUNT
130 'Y=COL WIDTH OF LINE
Z=LINE COUNTER
140 'X=TOTAL CHAR COUNT
OF=OVERFLOW OF CHAR TO BE CARRIED TO
NEXT LINE
150 'READS A SEQUENTIAL FILE FROM .DOC
OR DESKMATE TEXT AND
155 'RIGHT JUSTIFY AND SAVES TO NEW .DOC
FILE FOR FURTHER EDITING.
160 '
170 '
180 CLS:
PRINT "LINE WIDTH IS 70 CHAR.. CHANGE?
Y/N :"
185 INPUT ANS$:IF ANS$="Y" THEN
PRINT"EDIT 250 AND CHANGE Y TO DESIRED
LENGTH."
190 PRINT LW$"WHAT /ASC FILE DO YOU
WISH TO RIGHT JUSTIFY? :"
200 INPUT LW$:LOW$=LW$+"/ASC"
210 PRINT RJ$"NAME OF OUTPUT /ASC FILE? :"
220 INPUT RJ$:RTJ$=RJ$+"/ASC"
230 'LPRINT CHR$(27);CHR$(33):
LPRINT CHR$(27);CHR$(17)
240 INDENT=0:'***ADD INDENT FEATURE
LATER FOR 5 SPACES
250 Y=70:Z=1:S=1:
'REM Y=LINE LENGTH AND Z=LINE COUNTER.
S=A$(S)DTSET END IN SPACE
260 X=1:TB=1:SP=1:
'TB reflects cursor POSITION,X TOTAL CHAR
COUNT,SP SPACE COUNT
270 DIM A$(Y+1):DIM B$(30):SP=1:
'A$=CHAR READ,LSC=LINE SPACE COUNT,
SP=SPACE
280 '****WIDEST PRINT LINE "SET Y" IS 70
JUST AS IN DESKMATE TEXT***"
290 '
300 OPEN "I",#1,LOW$
305 OPEN "O",#2,RTJ$
310 A$(TB)=INPUT$(1,#1):OF$=OF$+A$(TB)
320 IF A$(TB)=" " THEN GOTO 440
340 IF A$(TB)=CHR$(13) THEN GOTO 450
345 IF EOF (1) THEN 365
350 IF TB=Y THEN GOTO 500
360 TB=TB+1:X=X+1:GOTO 310
```

```
365 PRINT TAB(1),C$
370 PRINT #2,C$:CLOSE 1:PRINT:
PRINT "FILE CLOSING":PRINT"#LINES="Z
380 PRINT"#CHAR="X:END
390 C=(LEN(C$)-1):C$=LEFT$(C$,C)
395 PRINT #2, C$
400 PRINT TAB(1) C$;:C$="":Z=Z+1:S=1:
PRINT TAB(80)" "
410 IF LEN(OF$)>0 THEN TB=LEN(OF$)+1:ELSE
TB=1
420 GOTO 310:'PRINT TAB(70) LEN(OF$)" "X
430 '
440 B$(S)=OF$:C$=C$+B$(S):S=S+1:OF$="":
GOTO 345
450 C=(LEN(OF$)-1):OF$=LEFT$(OF$,C)
455 PRINT C$;OF$;:PRINT #2,C$;OF$;
460 OF$="":TB=0:C$="":Z=Z+1:S=1:GOTO 310
470 '***IF LINE NOT SKIPPED AFTER CAR.RET.
THAN LAST LINE OF PARA RUNS OVER***
480 '***IF NO PRINT TAB(NN) LINE 330, LINE
AFTER C/R OVERLAPS PRIOR LINE**
490 '
500 IF LEN(OF$)=0 THEN GOTO 390
510 S=S-1:C$="":LO=LEN(OF$)
520 S2=INT(LO/S):M=LO MOD S:
'PRINT TAB(55)TB" "LO" "S2" "M" "S
530 '
540 LS=S*S2+M
550 FOR SC=1 TO S
560 IF LS>0 THEN B$(SC)=B$(SC)+" "
570 IF LS>S-1 THEN  B$(SC)=B$(SC)+" "
580 LS=LS-1
590 C$=C$+B$(SC):NEXT SC
600 'PRINT"Z"
610 GOTO 390
```



BEAT THE GAME

by Daniel Myers



SEASTALKER

As you sit quietly at the workbench in your research laboratory, you're startled into action by the sound of the videophone alarm bell. You'd better act quickly, because your buddy Tip Randall is raising the roof. The first thing to do is turn on the videophone. As soon as you do that, though, you realize that the picture is fuzzy. That's easy to correct; simply adjust the videophone. There is Commander Zoe Bly, looking worried, and telling you about an urgent problem at the undersea Aquadome. You'd better pick up the microphone, then turn it on.

After asking Bly about the problem, question her about the monster she's seen. Bly is sounding ever more desperate, so tell her goodbye. Suddenly, however, something's wrong with the videophone, and your score drops by 3 points!

Now is the time to go to the Computestor for a clue. First, turn off the microphone and drop the microphone onto the workbench. Then, head for the Computestor and turn it on. Since the machine is now ready for questions, ask it about the videophone. Hmmmm...the problem could be one of many, but you suspect that something may be wrong with the electrical panel. The panel is just down the hallway, so go to the panel, and examine it. Well, well, apparently the circuit breaker is open. By fixing the circuit breaker, you regain your 3 points. However, you are starting to wonder whether treachery is afoot here in the lab. It's time to have a chat with your assistant, Sharon Kemp.

Go to the office and confront Sharon with your suspicions. Her answers are evasive, and she seems very nervous. Since time is growing short, you decide to leave Sharon and head for your sub, the "Scimitar." Realizing that the sub won't start unless you have the atomic catalyst capsule, you first examine the work counter. There is the capsule, so you grab it and head for the Scimitar.

Once settled in the pilot's seat, with Tip nearby, you decide to check the sub for any problems. Pushing the test button gives you a positive readout, but you're still apprehensive. You will need to open the access panel in order to enter the sub's crawl space, but you don't have a tool. Maybe Tip has such an item? Tip comes through, handing you a Universal Tool. Open the access panel, and carefully crawl into the space. A check of the voltage regulator reveals that it is damaged. Use the tool to fix the regulator. Now all is A-OK, and you won't have any problems going full throttle to the Aquadome.

You're ready to get underway, so crawl out of the space, close the access panel, close the sub's hatch, and put the catalyst capsule into the reactor. After closing the reactor, you'll need to turn on the reactor and fill the docking tank with seawater. Once the tank is filled, turn on the engine, open the tank gate, then open the throttle. Push the joystick to the east, and you're off!

The surface of Frobton Bay isn't the safest spot around, so the first thing you need to do is set your depth to 5 meters and set the throttle to slow. You'll want to check the sonar occasionally to make sure you're not heading toward any obstacles. Your sequence of moves must be accurate to avoid destruction.

One quick way to reach the seawall opening is to follow these moves: Northeast, then three Norths, then Northeast again, then wait. The alarm bells may be ringing, but you'll safely avoid a submerged obstacle. Then, suddenly, an approaching ship is detected by the sonar. You'll have to stop waiting and set your depth to 15 meters to dive below the ship.

Wait again, and you'll chug right on through the seawall opening into the ocean.

Be sure to save the game here, since you won't want to cross Frobton Bay again! You can turn on the autopilot now, since the sub will head straight for the Aquadome. Because you fixed the voltage regulator, you can set the throttle to fast without overheating. Wait now, as you continue diving deeper and deeper. To check out an enormous whale, aim your searchlight to starboard. The trip will take a little while longer, so you might want to ask Tip about that magazine he's reading. A close study of a particular article in the magazine reveals that Dr. Jerome Thorpe (an Aquadome staff member) has succeeded in creating mutant sea creatures. Further, Thorpe announces in the article that he plans to marry your lab assistant, Sharon Kemp! You're beginning to understand who's behind the attack on the Aquadome, and you're even more anxious to arrive.

Wait a while longer, and then, as you near the structure, your sonarphone rings. It's Commander Bly, asking to speak privately with you when you arrive. You wait a few more turns, and the sub slows to a stop in the docking tank. Open the throttle to slide into the cradle. You wait while the water in the tank empties, and you save the game again.

Before opening the hatch and exiting the sub, you pick up the emergency oxgyen gear...just in case. Leave the Scimitar and head straight for the Aquadome's Reception Area where Bly and her crew await you. Greet them, and then take a quick look around. Your explorations are interrupted by a sudden realization that something is wrong with the air supply.

Quickly using the oxygen gear you so intelligently brought with you, head for the Dome Center. Commander Bly and several crew members are gasping for breath, so time is short. Use the universal tool to open the access door to the air supply assembly. Instantly noticing that something has been unscrewed from an important cylinder, you pick up the object. It is an electrolyte relay. Put the relay into the cylinder, and close the access door. Your efforts are successful, and the air supply is now functioning properly.

As you return to the Reception Area, you observe Doc Horvak with Bly's oxygen gear. You're suspicious, so when Bly ask you to accompany her to the office, you go with her. She volunteers some interesting information: She suspects sabotage in the Aquadome and shows you certain evidence. The evidence consists of a black box which you open and examine. This device could be used to interfere with the Aquadome's sonar, and Tip has an idea about how to trap the saboteur.

Go to the Storage Room with Tip and discuss his idea. Before you reach the storage area, you notice the special Fram Bolt Wrench lying under Bly's desk. Realizing that the wrench must have been used to tamper with the air supply, you show it to Doc Horvak. His reaction proves most interesting.

Now you need to do some serious thinking. Conversations with various crew members will assist you in your search for the traitor. Ask everyone about everyone else, check the locker in the men's dorm, set the black box onto the sonar, and observe everyone's behavior.

Commander Bly will offer to supply you with a bazooka so that you can hunt the monster (the "Snark"). Get that from her and have Tip install it on the sub's extensor claw. Find Doc Horvak and show him the magazine article about Thorpe. Doc will come up with some interesting conclusions, and will offer to prepare a special tranquilizer gun for you. Get the dart gun and have Tip install that as well.

During your explorations and conversations, Mick Antrim will check out the Scimitar then return and ask you whether you'd like to have an Emergency Survival Unit installed in the sub. You agree, then poke around a while longer until the unit is in place. It's time to think about improving your navigation and sonar -- the Snark will be difficult to capture or kill. You ask Tip about installing a fine grid and a fine throttle control in the sub, and he agrees to do so.

You're about ready to head out into the ocean again, but you still haven't come to a firm conclusion about who the Aquadome traitor is. Once in your pilot's seat, however, you notice that the survival unit installed by Amy and Bill is equipped with a nasty looking syringe. Grabbing the syringe, you head for Doc Horvak and ask him to analyze it. His analysis reveals that the hypo is filled with arsenic! You'd better confront Amy and Bill with this evidence before you do anything else.

The instant you show the syringe to Bill, he turns and runs away. He's heading for the sub, and you race to the office to view his actions on the station monitor. As you watch Bill climb down the inside ladder of the docking tank, you realize you have only seconds to trap him. You quickly turn off the docking tank electricity so Bill can't open the gate. He knows he can't get out now, so he surrenders.

You turn the electricity back on, and leave the office.

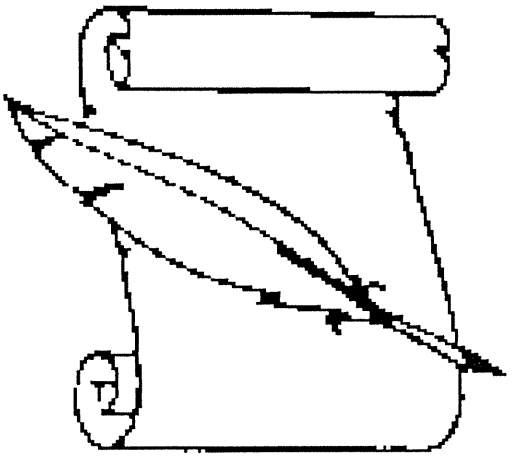
Cheers follow you as you head back to the Scimitar. After filling the docking tank with water, you turn on the engine and open the gate. Turning the joystick to the South, you open the throttle. Save the game, and head out into the ocean.

You're finally ready to confront the Snark and, perhaps, the evil Dr. Thorpe. Exit the Aquadome's docking tank by going South, then set the throttle to medium. Turn Southeast and wait until you reach the Snark and the Sea Cat (piloted by Dr. Thorpe). Thorpe will taunt you with his power, and admit his plan to wreck the Aquadome.

Suddenly, Thorpe's transmission breaks off, and Sharon Kemp begins to speak to you. She explains how she only went along with Thorpe to try to trap him, and she's ready to help you capture the Snark. Sharon has a lot of interesting things to tell you, but you don't have time to talk to her right now. The Snark is moving quickly toward the Aquadome, 9-ady to batter it to bits.

Here is one method you can use to put the Sea Cat out of commission before Thorpe has a chance to attack you: East twice, then check your sonar to make sure you're in position. Set throttle to slow, then turn South. Head Northwest four times. Oh oh! Dr. Thorpe has recovered consciousness and his voice is crackling over the sonarphone. Ignore him, and head Northwest twice more. The sub will be just to the East of the Sea Cat, so, all on one line, enter the following command: West then aim bazooka at power pod then shoot power pod with bazooka.

There! You've done it! The Sea Cat is out of commission and Thorpe's out cold again. Sharon guides the Snark to its hidden cavern so that you can safely study it later. You've completed your mission and saved the Aquadome!



C Programming Tutorial

Part 3

By J.F.R. "Frank" Slinkman

When working with arrays in a BASIC program, you don't have to worry about the size of array elements.

It doesn't matter if it's an integer array, or an array of single or double precision numbers, if you want to access array element number 5, you just put the value 5 in the subscript, and BASIC fetches the desired value for you.

The same is true in C, of course, and in every other programming language I know anything about.

But in C, you can access array another way -- through the use of pointers.

The pointer method is a FAR more efficient way to step sequentially through elements of an array than, for example, a "for" loop.

In "prog04.c" in Part 2 of this series, we used a "for" loop governed by the length of the string to look at each character in sequence.

A much better way to do this is to declare a "pointer-to-char" variable, initialize it with the address of the start of the string, and keep bumping it's value until it points to the character after the end of the string.

This is easy to do with strings since, if you'll recall, every string in C ends with a null character.

In other words, we just keep bumping the pointer until it points to a zero character.

So, load prog04/ccc into your text editor, and make the following changes:

1. Change the name of the program to "prog04a.c."

In the count_chars() function:

2. eliminate the "i" and "length" variables;

3. add a new variable declaration line, namely
   char *ptr;

4. change the line
   length = strlen(inbuf);
   to
   ptr = inbuf;

5. change the line
   for (i = 0; i < length; i++)
   to
   while (c = *ptr++)

6. remove the line:
   c = inbuf[i];

remembering that the left brace must NOT be removed.

After you have made these changes, the first 6 lines inside the function should be:

```
int    c, alpha, numeric, space, punct;
char   *ptr;

alpha = numeric = space = punct = 0;
ptr = inbuf;

while ( c = *ptr++ )
{  if ( isalpha( c ) ) alpha++;
```

Let's look at the statement "ptr = inbuf;".

"Inbuf" is the RAM address of the first character of the string input from the keyboard. This statement merely copies that address to "ptr" so it, too, points to the same character.

The "while (c = *ptr++)" does several things.

First, it picks up the object at "ptr" (i.e., the character being pointed to by "ptr"), and assigns that value to the variable "c."

Second, it increments "ptr" to make it point to the next character in the string.

Third, it evaluates the new value of "c," and makes a decision based on its value.

If "c" is a valid string character, the result will be TRUE (non-zero), and the "while" loop's compound statement will be executed. But if "c" picks up the null character at the end of the string, the result will be FALSE (zero), and the loop will be

exited.

It's important to understand the meaning of "*ptr++." As I stated above, it references the data item being pointed to by "ptr," and then increments "ptr."

It could also be written "*(ptr++)." In other words, it does NOT increment the VALUE being picked up. It increments the POINTER.

By contrast, "(*ptr)++" would increment the value being picked up, but not change the value of "ptr."

And "(*ptr++)++" would increment both the pointer and the value.

Now save "prog04a/ccc," compile and run it. You will see it runs identically to the original version, but is both smaller and faster. (Well, you may not notice the increase in speed in a process this short and simple, but take my word for it -- it's faster.)

There is one aspect to the operation of "ptr++" which might be easier for BASIC programmers to get used to than for assembly language programmers.

And that is the idea of the "scaling" of pointers. In this case, because "ptr" was declared a pointer-to-char, each operation of "ptr++" adds one to it's actual value.

But what if "ptr" was declared a pointer-to-int, or pointer-to-long, or pointer-to-double, the values of which are stored in 2, 4 and 8 bytes, respectively?

In these cases, unlike pointers in assembler, "ptr++" causes "ptr" to point to the next element of the array -- NOT to the next byte in RAM! In other words, it will add 2 to the actual value for ints, 4 for longs or floats, and 8 for doubles.

To confirm this, type in, compile and run the following program:

```
/* prog05.c */

#include   <stdio.h>

char   c_array[10], *c_ptr;
int    i_array[10], *i_ptr;
long   l_array[10], *l_ptr;
float  f_array[10], *f_ptr;
double d_array[10], *d_ptr;

main()
{  int    i;

   c_ptr = c_array;
   i_ptr = i_array;
   l_ptr = l_array;
   f_ptr = f_array;
   d_ptr = d_array;

   printf( "\x1c\x1f" );
   puts( "count\tchar\tint\tlong\tfloat\
   tdouble" );
   for( i = 0; i < 10; )
      printf( "%2d%8u%8u%8u%8u%8u\n",
         i++,
         c_ptr++,
         i_ptr++,
         l_ptr++,
         f_ptr++,
         d_ptr++ );
}
```

Did you notice where the "i++" was placed -- OUTSIDE the parentheses associated with the "for" statement? Not only does this demonstrate that this can be done, but in this case it's the most efficient way to do it.

Also notice that while we declared the five different arrays, we never initialized them. That is, the number of bytes of RAM needed to hold ten values of each type of data were allocated, but we didn't put any meaningful data into that RAM space.

But it doesn't matter, because this program is only interested in the behavior of pointers, not in the contents of the arrays.

Prog05.c displays a chart on the monitor screen. Except for the header and the "count" column, it displays the RAM addresses, in decimal, of each of the ten elements of each of the five arrays. It gets these values from pointers to the elements.

Note that, going down each column, each value in the "char" column is one greater than the value above it. Each value in the "int" column is two bytes greater; each value in the "long" column is four bytes greater, etc. Proof positive, in other words, of the automatic scaling of pointers.

Now, just for fun, turn your printer on and, from LS-DOS Ready, invoke the program as follows:

```
prog05 >*pr
```

Instead of the chart being printed on our

screen, it has been routed to the "*pr" device, through what is called "standard I/O redirection."

If you were to invoke the program via

```
prog05 >chart/asc:0
```

the chart would be written to the named disk file.

Try redirecting the output to a disk file, and then LIST the file. You should see the chart on your screen identically to what you saw when you ran the program without redirection.

Now use your text editor to create a one-line string (not to exceed 79 characters + CR), which must be terminated by a carriage return. Save it out to a file named "string/asc."

Now invoke "prog04a" as follows:

```
prog04a <string/asc
```

In this case, the data from string/asc was read from disk and used as the data for the program instead of input from the keyboard.

The string won't be displayed as it was when obtained from the keyboard, but the program still processes the data just fine.

Refer to your compiler manual (page 1-8 for ProMC) for a more complete discussion of standard I/O redirection.

I'll leave it as an "exercise for the reader" (don't you just hate those?) to alter prog04a.c in such a way that it will display the string before it displays the report.

Another exercise for the reader (especially readers who are "into" assembly language and think in hex) is to alter prog05.c to print the pointers in hex, rather than decimal.

O.K. It's now time to do a little review.

There are ten "statements" in the C language:

1 "Break" -- causes immediate exit from within a "for," "while," or "do" loop, or from a "switch" statement clause. We've covered this pretty well, except for the switch statement part.

2 "Continue" -- this statement is used in loops, and causes any statements which follow it to be skipped. The loop then goes to its next iteration as normal. We haven't covered this yet, but will soon.

3 "Do" -- creates a program loop. We've covered this.

4 "For" -- creates a program loop, also covered.

5 "Goto" -- yes, despite the influence of the advocates of structured programming, you can do a "goto" in C. We have not covered this yet.

6 "If" -- covered.

7 The "null" statement -- covered.

8 "Return" -- covered.

9 "Switch" -- we'll get to that shortly.

10 "While" -- covered.

In addition, some texts classify the ability to combine multiple statements as an 11th statement:

11 Compound statements -- a number of program statements combined into one through the use of braces. These are also known as "blocks" of code. We've pretty much covered these, except for the fact that blocks can also have their own, unique variables, which are accessible only within that block.

Not bad. We haven't even finished the third lesson, and we've already covered 73% of the statements. So let's see if we can't kill the last three birds -- continue, goto, and switch -- with one stone.

We're going to go to Las Vegas and play craps. Specifically, we're going to test an admittedly stupid craps system: namely to place bet all the numbers on the comeout, leave the bets up for three rolls, then take them down until the start of the next pass.

While it's a silly system, it's perfect for demonstrating the use of the three remaining statements.

But before we start coding, you need to learn a little about craps. A "pass" is a number of rolls of the dice. If the shooter's first roll is 2, 3, 7, 11 or 12, the pass is over. Seven or 11 are winners; 2, 3 and 12 are "craps" -- losers.

If he rolls 4, 5, 6, 8, 9 or 10, that number becomes his "point."

From then on, he keeps rolling until he rolls either a 7 or his point. Seven is a loser. The point is a winner. Either of these rolls ends the pass.

You can "place bet" any or all of the numbers 4, 5, 6, 8, 9 and 10. If the number comes up, the bet wins. If the shooter rolls a seven, the bet loses. The 4 and 10 pay 9:5. The 5 and 9 pay 7:5, and the 6 and 8 pay 7:6.

```
/* prog06.c */

#include   <stdio.h>
#include   <time.h>

#option INLIB
#option ARGS       OFF
#option REDIRECT   OFF
#option FIXBUFS    ON
#option MAXFILES   0

#define BREAK  0x80

void   play_craps();

long   money = 0L;
int    dummy[7], *place;

main()
{  int    i;

   memset( dummy, 0, sizeof dummy );
   place = dummy - 4;
   srand( (int)time( NULL ) );

   for ( i = 4; i <= 10; i++ )
   {  if ( i == 7 )
         continue;
      place[i] = ( i == 6 || i == 8 ) ? 6 : 5;
   }

   while ( rollem() != 7 )
      ;

   printf( "\x1c\x1f" );

   play_craps();
}

void play_craps()
{
   int    roll, action, c_point, roll_ctr;
   long   pass_ctr = 0L;

new_pass:
   cursor( 24,6 );
   printf( "After %ld passes, you have $%ld\x1e",
         pass_ctr++, money );
   cursor( 17,8 );
   puts( "Press BREAK to exit, any other key to \
continue" );

   if ( getc( stdin ) == BREAK )
      return;

   cursor( 0,11 );      putc( '\x1f', stdout );
   money -= 32L;
   roll_ctr = 0;

   action = pay_place( roll = rollem() );
   if ( action == -1 )
      goto new_pass;
   else if ( action == -2 )
   {  money += 32;
      goto new_pass;
   }
   else
      c_point = roll;

new_roll:
   if ( ++roll_ctr >= 3 )
   {  money += 32L;
      printf( "  **" );
      do
         roll = rollem();
      while ( roll != 7 && roll != c_point);
      goto new_pass;
   }

   action = pay_place( roll = rollem() );
   if ( action == -1 )
      goto new_pass;
   else if ( action == -2 )
      goto new_roll;

   if ( roll == c_point )
   {  money += 32L;
      goto new_pass;
   }
   else
      goto new_roll;
}

int pay_place( roll )
int roll;
{
   int    retcode = 0;

   switch ( roll )
   {  case 4: case 10:
         money += place[roll] * 9 / 5;
         break;
      case 5: case 9:
```

```
            money += place[roll] * 7 / 5;
            break;
         case 6: case 8:
            money += place[roll] * 7 / 6;
            break;
         case 7:
            retcode = -1;
            break;
         default:
            retcode = -2;
      }
      return retcode;
}

int rollem()
{
      int    die1, die2, total;

      die1 = rand() % 6;
      die2 = rand() % 6;
      printf( "%4d", total = die1 + die2 + 2 );
      return total;
}
```

O.K. Lots of new stuff to cover here.

First are a bunch of "preprocessor directives," namely the #include, #option and #define lines.

Note we have included the time.h header. This is because the time() function is not declared in stdio.h.

It's the responsibility of the programmer to make sure all the library functions he's using are given forward "external" declarations, either those appearing in a header file, or by a line like:

```
extern long time();
```

which would normally be either be grouped with the other forward declarations (i.e., before the first executable code in the program), or appear before the first line of executable code in the function in which the extern function is used (main(), in this case).

The #include and #define are standard C, and thus are fully portable. The #define is straight a text substitution macro. In this case, it tells the compiler, "from now on, whenever you see the string, 'BREAK,' replace it with the string, '0x80'."

The "0x80" is the C way of expressing a hexadecimal value, in this case decimal 128, assembler 80H, or BASIC &H80. However you express it, it's the value of the "BREAK" key.

The five #option lines are not standard, and thus are not portable. However, they can be very useful to ProMC users.

"#option INLIB" instructs the compiler to search the IN/REL library for some special, non-standard functions. In this case, we want to use cursor().

While most versions of C have this function, it isn't part of the defined standard libraries, which means you can't assume it exists on other systems.

"#option ARGS OFF" tells the compiler there will be no command line arguments required to invoke the program; so the code to handle them can be omitted, making your final /CMD program smaller.

"#option REDIRECT OFF" tells the compiler to omit the code to handle standard I/O redirection, which also makes the final /CMD program smaller.

"#option FIXBUFS ON" has to do with memory management and allocation. C provides ways to allocate and deallocate RAM (roughly similar to defining and erasing arrays in BASIC). If the program doesn't need this feature, the code supporting it can be omitted, making the final /CMD file smaller.

"#option MAXFILES 0," which operates in conjunction with FIXBUFS, specifies the number of file buffers your program needs (in addition to the three standard files -- stdin, stdout, stderr -- which are always present). ProMC provides for 13 additional file buffers. If FIXBUFS is ON, then RAM will be allocated for all 13 whether you need them or not. Thus, defining MAXFILES to the actual number of file buffers needed will free up quite a bit of RAM for use by your program.

In short, the last 4 of the options are used to make the final /CMD program as small as possible and give the program access to as much RAM as possible.

Even though they're non-standard C, I make it a habit of using all that apply to the program I'm writing. After all, they're easy to remove if I ever need to make the code portable.

Now notice the declaration and initialization of the variable "money." It's declared to be a long int, and is initialized with the value "0L," not just zero.

Just plain zero ("0") would work, but it would require the compiler to take the 2-byte int zero and convert it to a 4-byte long zero before storing it.

The "0L" skips the conversion; so is more efficient.

Next we declare an array of seven ints, and declare "place" to be a pointer-to-int. We're going to do some tricky things with "place" later.

The first thing we do in the main() function is to zero out the "dummy" array, using the "sizeof" operator. "Sizeof" is very handy and useful.

"Sizeof" can be used to get the size, in bytes, of every data type used by the system. This includes the standard ones like char, int, and double, but also the ones you define, like arrays.

In this case, it knows how big the whole dummy[] array is, and the size of each element of the array; so we can use it in the memset() function to write bytes of zero to the entire array.

If we had used "sizeof(dummy)," it would take on the value of two, the size in bytes of one element (a short int) of the array. Thus, if your program needs to know how many elements there are in the dummy[] array, it could find out via:

```
elements = sizeof dummy / sizeof(dummy);
```

Instead of memset(), we could have used the ProMC function zero(), a la:

```
zero( dummy, sizeof dummy );
```

but this is non-standard. It's best to adopt the habit of avoiding the non-standard functions as much as possible, especially when there are perfectly good standard functions which do the same job. Unlike the #options discussed above, the use of non-standard functions isn't that easy to change if you want to make the code portable.

The next statement is a bit unusual. What "place = dummy - 4" means is "make 'place' equal to the address of element number minus four of 'dummy'."

This is an excellent example of the use of the automatic scaling of arrays. At first glance, it looks like the number 4 is being subtracted from "dummy."

But, because "dummy" is a pointer to an array of ints, what is actually being subtracted is 4 times the size of an int.

The statement could also have been written:

```
place = &dummy[-4];
```

The "&" is the "address of" operator. It tells the compiler to get the address of the data, rather than get the data itself.

Of course, there is no element number -4 in any array. We're just imagining it. But this statement sets up a correlation between "dummy" and "place" so that place[4] is actually dummy[0]; place[10] is actually dummy[6]; etc.

We have no need of array elements 0 through 3; so why waste RAM by allocating it to store data that will never exist?

The next statement uses two standard library functions, srand() and time(), to reseed the random number generator. "Time(NULL)" returns the UNIX time, which is the number of seconds since 00:00:00, January 1, 1970.

Time() returns a long (32-bit) int, which we truncate to a short (16-bit) int by throwing away the top 16 bits by casting the return value to "int." This value is then used to seed the random number generator via srand().

The "for" loop, by using values of "place" from 4 to 10, is actually talking about dummy[] elements 0 through 6, as discussed above.

We can't place bet the seven; so we skip this value through the use of the "continue" statement. In this case, if the value of "i" is 7, the rest of the loop (one statement in this case) will be skipped.

Otherwise, the rest of the loop will be executed.

The next statement is a variation of the "if-then-else" construction. It's called the "conditional operator." The general form of this operator is:

```
variable = (expression) ? a : b;
```

Both "a" and "b" can be anything that has or returns a value, including variables, numbers and function calls, or even another conditional operator.

In pseudo code, the logic is:

```
1. is "expression" TRUE?
   YES:
      assign "a" to "variable"
   NO:
      assign "b" to "variable"
```

This particular statement is IDENTICAL to:

```
if ( i == 6 || i == 8 )
```

```
        place[i] = 6;
    else
        place[i] = 5;
```

Next, we use while and null statements to keep calling our rollem() function until it returns a 7. In craps, every "pass" ends with a roll of 7; and we don't want to start playing in the middle of a pass.

Next, we clear the screen and, now that all the preparations are done, we're ready to play; so we call play_craps().

After the declarations you'll see, on the left margin, followed by a colon, a program "label." This location in the program is the start of the processing of a new craps "pass." Later in the program, we'll use "goto new_pass" statements, and "goto" statements can only route program flow to a label.

First, we print a couple of messages (note the use of cursor() to position them on the screen) and use the standard library getc() function to get one keystroke from the keyboard (stdin).

For your information, ProMC has a non-standard function, getchar(), which is identical to getc(stdin).

Whether you use getc() or getchar(), the function waits for a key to be pressed, and returns its value.

Once the value is returned, this code tests it. If it was the break key, the play_craps() function terminates and returns to its caller.

(This getc(stdin) behavior is unusual. On MeSsDOS machines, for example, the user has to press the RETURN key before getc(stdin) will return; so you can't program things like "Press any key to continue" prompts. Score another one for the TRS-80!)

Next we position the cursor in column 0 of line 11, and clear the screen from there to the end of frame by sending chr$(31) to the screen via putc().

Then we subtract the total of the place bets from "money" and set "roll_ctr" to zero.

Now we roll the dice by calling our rollem() function, placing the return code in "roll", and sending the value of "roll" to our pay_place() function.

The value returned by pay_place is loaded into "action." Pay_place() returns zero if it's argument is 4, 5, 6, 8, 9 or 10; -1 if the argument is 7; or -2 if it is sent any other value.

Thus, if "action" has the value -1, then the roll was 7, causing the place bets to lose, and the pass is over; so "goto new_pass" is executed.

If "action" is -2, then one of 2, 3, 11 or 12 was rolled, which does not affect place bets, but do end the pass; so the total amount of the bets is added back to "money" before going to "new_pass."

Otherwise, the roll must be one of 4, 5, 6, 8, 9 or 10; which becomes the shooter's point; so "c_point" is loaded with the value of "roll."

Why did we name the variable name "c_point" instead of just "point?" Well, because we #included IN/REL, which defines the function point(), which returns the state of a screen pixel.

Avoiding variable names which are the same as function names avoids the possibility of conflicts which could confuse the compiler, resulting in error messages, and/or create errors in the final program which can be very hard to track down. Now you know why most functions have weird names -- names you would be unlikely to choose as names for variables or your own functions.

As the next label, "new_roll," indicates, it's time for the shooter to roll the dice again. Labels MUST be on the left margin and MUST be followed by a colon.

If the pre-incremented value of "roll_ctr" has reached three we, in effect, take the place bets down by adding back the total of those bets to "money," printing a symbol on the screen indicating the bets have been taken down, wait for rollem() to return either 7 or the shooter's point, indicating the end of the pass. Once this happens, program control goes to the label "new_pass" by way of the "goto" statement.

If roll_ctr is not yet 3, we roll the dice again and test the result via the pay_place() function.

If "action" is -1, a 7 was rolled, the place best lose, and the pass is over.

If "action" is -2, 11 or craps was rolled, which doesn't affect place bets; so the shooter rolls again.

Otherwise, the roll was one of the place numbers, and is checked to see if the shooter made his point. If so, the place bets are taken down, and the pass is over.

If not, the shooter rolls again.

Obviously, this function could have been written in structured programming style without the much maligned "goto" statement.

I'll leave it as another obnoxious "exercise for the reader" to rewrite it without the accursed "goto's."

Now we get to the pay_place() function, which is basically a "switch" statement with a return code.

First we initialize "retcode" to it's most likely value, zero.

"Switch" is the only statement which REQUIRES the use of a compound statement. The argument for "switch" must be a short int.

Within the "switch" statement's compound statement, we use "case" clauses to call for desired processing. In this example, "case 4: case 10:" means "do the following if "roll" equals 4 or 10.

If you'll remember, place bets on the 4 and 10 pay 9:5; so we add 9/5ths of the bet to "money," and exit the "switch" statement via the "break" statement.

Similarly, bets on the 5 and 9 are paid 7:5, and bets on the 6 and 8 are paid 7:6. In the case that "roll" equals 7, "retcode" is set to -1. In all other cases, "retcode" is set to -2.

Notice the "break" statements. They cause program control break completely out of the "switch" statement (i.e., to the statement immediately after the "switch" statement's closing brace). Without the "break" in the "case 4: case 10:" clause, for example, the code in the "case 5: case 9:" clause would be executed. Notice also that the last "case" doesn't require a "break," since program control from there falls through to the statement after the "switch" statement anyway.

Obviously, in situations where you WANT the code in the next "case" clause executed, you would DELIBERATELY omit the "break" statement.

Notice one clause can handle one or more cases, and you can use "default:" to catch the non-specified cases.

Anyway, once "switch" has paid off the bets or assigned a new value to "retcode," it returns "retcode" to the caller.

The rollem() function uses the random number generator to roll two dice.

For each die, it generates a random number via the standard rand() function, which returns a short int in the range 0 to 32767. We then use the modulo division operator ("%"), to convert that to a value in the range 0 to 5.

Except for the fact that rand() can return a value of zero and BASIC's RND() command cannot, the line:

```
die1 = rand() % 6;
```

is identical to the BASIC

```
DIE1 = RND(32767) MOD 6
```

Then we take the total of the two dice, plus 2 to compensate for the fact that the range of each should be 1 to 6 instead of 0 to 5, and put that value in the "total" variable, which we then display on the screen.

Finally, we return "total" to the calling routine.

Type in prog06.c, and compile it via

```
mc (n="prog06",o )
```

Now, before running it, do a DIR PROG06, and look at the difference in size between PROG06/ASM and PROG06/OPT. As you can see, ProMC's optimizer, MCOPT, does a rather good job of making assembler code more efficient.

Normally, because it can take a long time to optimize even a moderately large program, what I do with a newly written program is compile it without optimization, just to see if it works right. Only when I've got it perfect will I:

```
mcopt progname:d
```

and then

```
mras mc +i=progname/opt +o=progname -nl

mlink progname -n=:d -e
```

and remove the /TOK, /ASM, /OPT, and /REL files.

Well, that about does it for this issue. But be sure to tune in two months from now for the next exciting episode, when we'll explore the data organization techniques built into the C language, namely "structs" and "unions."

PROGRAMMING TIDBITS

A Short Boole Session OR The Imps Raise Their Ugly Heads Again AND How To Put Them Back Where They Came From

A year ago ("Basic Imps", TRSTimes 6.2) we tried to explore all the "logical operators" of Model 4 BASIC, including the obscure IMP. One of the conclusions of that review was that all programming can be done just fine with the simple AND, OR and NOT. The Mod-4 additions: EQV, IMP and XOR are interesting and sometimes can make the code more "elegant", but that's about all. Case dismissed.

Recently, however, the "imps" surfaced again in a letter from Henry Herrdegen and in the Editor's response to it (TRSTimes 7.3). The Editor swiftly unmasked the mysterious XNOR function on Henry's calculator: it does exactly the same thing as EQV in Model 4. But why there is no IMP on the calculator? On this innocent question dangle some 2000 years of Western thought.

As noted in "Basic Imps", the IMP operator is the only one for which the order of operands can make a difference. For example "0 IMP 1" returns 1, but "1 IMP 0" returns 0. Other operators never care about the order of operands: "1 OR 0" returns 1, and so does "0 OR 1". You can check it out for AND, XOR, and so on.

Because of the commutative and associative laws of mathematics, the order of operands is irrelevant in most cases. A calculator designed for scientific math would have very little use for the IMP operator, and that's probably why it is not included.

Not so in human logic: the "implication" is one of the basic tests of our reasoning. Consider this statement: "the man never drinks, so he is sober today" (here the colloquial "so" stands for the more formal "implies that"). There are two factual components (operands) of our sentence:

(a) man never drinks
(b) he is sober today

To verify our reasoning we can examine all possible cases of the implication's truth table which looks like this:

(a)	(b)	(a) IMP (b)
0	0	1
0	1	1
1	0	0
1	1	1

Substituting the true and false instances of the facts (a) and (b) into the 1's (true) and 0's (false) of the table we get:

(a)	(b)
sometimes drinks	drunk today
sometimes drinks	sober today
never drinks	drunk today
never drinks	sober today

Obviously all combinations can be true exept the third: it is not possible in real life to get drunk without ever drinking. Thus our reasoning is correct: it fits the truth table. But if we reverse the terms ("the man is sober today, so he never drinks") then our reasoning fails at once. It is perfectly possible for the first fact to be true (sober today) while the second is false (sometimes drinks), and such outcome is prohibited by the IMP truth table.

This "uni-directional" quality of implication, compared with the "bi-directional" character of other logical operations, puzzled Western thinkers ever since Greeks invented philosophy. It was a source of occasional bitter fights between various "schools" of logic until a Britisher named George Boole in the XIX-th century extracted all the "imps" from the messy jungle of human language and organized them into a complete system of "Boolean operators". As it turns out, even more "imps" exist than we have in Model 4 BASIC: there are altogether 16.

How come 16? Two operands, each true or false, can be arranged in 4 different ways in a "truth table" where "true" is shown as "1" and "false" as "0". Now, since each table has 4 possible outcomes, there are 16 ways to arrange these results, and thus 16 different Boolean operators. Some have names such

as AND, OR, etc, others have no names. Some are useful, some are not, but there are computer programming languages (for instance some variants of LISP) that provide a general "Boole" function capable of performing any of the 16 Boolean operations.

However, there is no reason to be intimidated by the sheer numbers of those "imps", because many of them are more "bull" than Boole. First of all, half of them are just opposites of the other half, obtained by the application of the "unary" operand NOT to the result. Take for example Henry's XNOR (or EQV as it is called in Model 4). It is simply the opposite of XOR, as we can see by comparing the "truth tables" of both:

Operands		XOR	XNOR/EQV
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

For example if "0 XOR 1" returns 1, then "NOT (0 XOR 1)" returns 0 which is the same as "0 XNOR 1" or "0 EQV 1". The EQV (short for "equivalent") is actually a better term, more expressive of what happens in this operation: the result is true when both operands are equal. Also, it would seem that NOT XOR should be properly called NXOR to be consistent with the names of other similar operators whose truth tables are negations of their "siblings", such as NAND (stands for NOT AND):

Operands		AND	NAND
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

The NAND truth table should be familiar to hardware buffs. It is used extensively in electronic circuits (the so-called "NAND gate" is a basic building block of our computer's RAM memory). In the same manner we can construct a NOR table by NOTing the OR table, etc.

Okay, so we have exposed half of the "imps" as being merely the NOT copies of the other half. That still leaves 8, but three of those are rather degenerate. A truth table where all results are "true" (1) regardless of the truth of the operands is called a "tautology" and obviously is useless: it doesn't tell us anything about the operands. The other two useless operators are when the result depends only on the truth of one of the operands, regardless of whether the other operand is true or not. In those two cases we simply ignore the irrelevant operand and that's that.

Now we are down to 5, but since IMP is directional, there are two possible arrangements of an IMP truth table, depending on the order of the operands:

Operands		IMP	Operands		IMP'
0	0	1	0	0	1
0	1	1	1	0	0
1	0	0	0	1	1
1	1	1	1	1	1

In the second IMP' the operand columns are the same as in the first, only their order is swapped. The logic is still the same in both cases, and thus only one IMP is relevant.

It seems we have now 4 useful operators, but why stop here? Don't forget the trusty NOT. By applying it to the first operand before ORing it with the second the same results are obtained as with the IMP:

(a)	NOT (a)	(b)	OR
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1

In BASIC this could be written as "(NOT a%) OR b%". This operation (politely called "dissolution of implication") hopefully kills the egregious IMP once and for all and leaves us with the only three essential operators:

Operands		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

But, you'll object, Christopher lover you said in "Basic Imps" that XOR is not really essential, so what gives? The answer is "both". In terms of fundamental logic these three operators (plus the NOT) are indeed necessary. As you look at the above table, there is an elegant symmetry. This symmetry is not accidental. It simply covers the three basic statements that can be made about two items of interest: both are true, both are false, or they are different. Actually in formal symbolic logic the EQV (the NOT counterpart of XOR) is used more often than XOR, but it's the same idea. In computing we prefer XOR because it is a handy "toggle" for bit manipulations: each time we XOR

any bit with 1, that bit flips from 0 to 1, and from 1 to 0.

But if we relax the rules a little and agree to accept expressions more complicated than just sneaking a NOT here and there, then we can also eliminate XOR. For example Model III BASIC does not have XOR, but we can define a function like this (all in one line):

```
DEF FNXR%(a%,b%)=
(a% OR b%) AND (NOT (a% AND b%))
```

and then instead of:

```
z% = x% XOR y%                    'Model 4
```

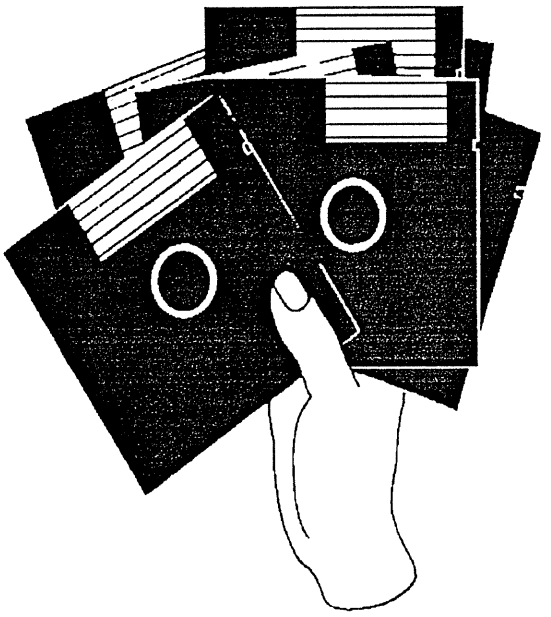
we can write:

```
z% = FNXR%(x%,y%)   'Model III
```

Similar functions in terms of AND, OR, NOT could be written for any of the 16 Boolean operators.

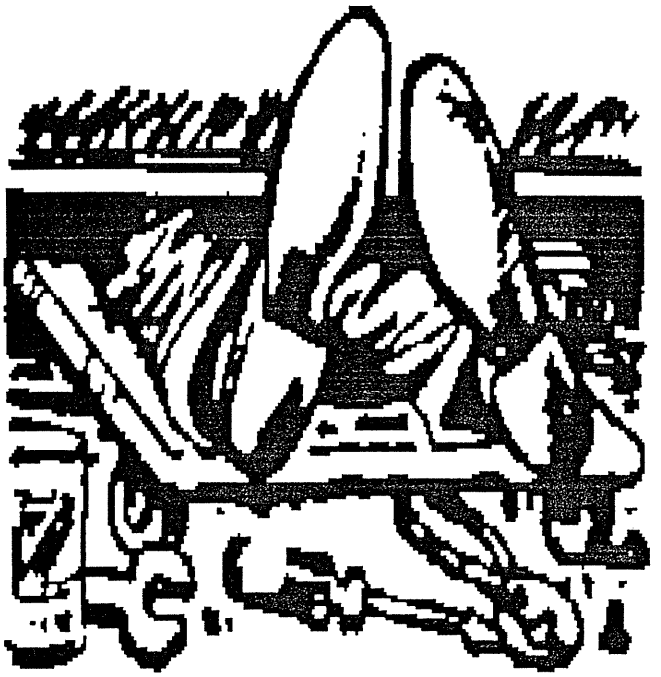
Actually it's possible to express any Boolean operator using only NOT plus AND, or NOT plus OR. But the formulas get so absurdly convoluted that the exercise is only of academic interest.

Anyway, next time any of the other "imps" come bothering you again, you'll know where they are coming from and how to get rid of them.



Some Hacking Reminiscences

by Roy T. Beck



What is a "hacker"?

Originally, a hacker was a clever, but reasonably honest computerist who could modify or create programs as needed to accomplish special functions, overcome machine or program deficiencies, and patch programs to solve problems the original programmer had failed to catch. (These latter problems were sometimes described as "undocumented features" when the users complained!)

As us old timers know, the term "hacker" has been greatly corrupted since it was first introduced. Nowadays, the term hacker is being applied to computerists who break into other people's systems via modems for the purpose of malicious mischief or outright theft.

In this article, I will describe some activities by both present day hackers, as reported in the newspress and other sources, and some along the original meaning of hacker, including some personal experiences.

Recently, a hacker name William Allen Danforth and his coconspirator Michael William Lazzarini have been found to have been operating as a merchandiser of stolen programs and pornographic images. The operation was run by the two employees of Lawrence Livermore National Laboratory at Livermore, California. One was the mastermind, the other was performing manual functions in the computer room to enable the disemination of porn and bootlegged commercial programs. According to published reports, over 90,000 pornagraphic images, some involving children, were stored on tapes, and were sold to unscrupulous buyers via the Internet system. Since graphic files are large, over 50 gigabytes of storage, paid for by you and me, the taxpayers, were involved. The Alameda County District Attorney is filing suit against the principal perpetrator, with a possible penalty of three years in prison and $10,000 fine for each of the two counts filed against him. Lazzarini, his partner is facing lesser penalties. The cache of materials was stored on tapes; Lazzarini was mounting and storing tapes at the direction of Danforth, the principal. Danforth has "resigned" from the Laboratory, and could not be reached by the L. A. Times reporter for comment.

When the Lab was tipped by the L. A. Times as to what was going on, the Lab launched a serious investigation, which was said to have cost an additional $13,000 to perform. More expense to us taxpayers.

I am sure many of you have read "The Cuckoo's Egg", which describes the activities of a pair of hackers, based in Germany, who broke into many US computer systems, looking for information and files of value to the East German government. These birds were utilizing the Internet system, which is an informal network, interconnecting numerous government and university computers.

Their targets were government and university computers containing data and programs of military or political value. The author of The Cuckoo's Egg was at the time employed at the University of California, Berkeley doing the necessary programming

and accounting to allocate the various costs of operating the UC system to the appropriate users. He became aware of the hackers because there were small discrepancies in the accounting results; his effort to account for the small discrepancies revealed the presence of the hackers. The author did some detective work to discover what was going on, and found the hackers had discovered various means of entry into the computers, including "trap door" entry into some operating systems, which in effect gave the hackers managerial control over the UC system. The hackers set up their own accounts and arranged to bypass the accounting routines in the OS, which for the most part made them invisible to the legitimate system managers. Since the system was large, with many users, the small discrepancies in telephone company charges vs the recorded accounts went unnoticed for a long time, the discrepancies being ignored, or thought to be the result of rounding errors, etc. The author basically decided on his own initiative to pin down these accounting discrepancies, and this led to his discovery of the presence of the hackers.

The book also reveals the difficulties in gaining cooperation between such agencies as the universities, the FBI, and others who should be concerned. It lead to great frustration, at times, on the part of the author. That book is recommended to all serious computerists.

Coincidentally, the author of the Cuckoo's Egg, Cliff Stoll, was formerly one of the system managers at the Livermore Lab where the recent hackers were doing their tricks. It would have made a great story if stoll could have detected and trapped the hackers abusing the Lab computers, but it didn't happen that way. Instead, a reporter from the L. A. Times discovered what was going on and tipped the Lab managers.

Many years ago, I heard a story about a programmer working in a NY City bank. His shtick was that he thought about the small errors due to rounding off when computing interest earned by the customers' savings account. He proceeded to create a secret account in the computer system to which all the fractions of cents of earned interest were credited. With the large bank business flow, this account mounted rapidly in size. The hacker then instructed the operating system to periodically write a check for the balance in the secret account, which was mailed to him in a roundabout fashion. He made a neat pile out of it until another system programmer had reason to review code in the OS, and tripped over this undocumented routine which credited the fractions of cents to the secret account. According to the story, the hacker was fired, but not charged with any crime, on the basis that the bank did not want to reveal the ease with which the hacker had penetrated the security features of the bank's system.

My niece is a systems programmer for a local bank, and she told me of an experience she had when she first went to work for the bank. She was hired as a replacement for another person. I gather the other person, the hacker, was let go for some undisclosed malfeasance. In any event, her first evening at the bank was interrupted by the previous employee dialing into the computer center via MODEM. I don't know if he actually interfered with the computer programs, but he had all the passwords required to allow him to do something serious. My niece realized the seriousness of the penetration, and tried to exclude the hacker. The problem was there were numerous MODEMs, each with separate phone numbers. Also, she could not find a way to exclude him in software, since his calls had all the proper passwords. She then began an effort to exclude him by disconnecting each MODEM he used. The problem here was that the modems were housed in enclosed racks, and she did not have keys available. But the racks had openings near the bottom for cable entry. My niece is very petite and was able to insert an arm through the cable entries to disconnect the cables at each MODEM. She spent an exciting evening disconnecting MODEMs to kill each penetration by the hacker! Of course, the passwords were changed the following day, and the hardware restored, but she had an exciting introduction to the bank's computer system.

Another case of inadvertent hacking was done by me, some years ago. I was then quite a novice and was using my Model I with the terminal program ST-80 by Lance Micklaus, and was accessing a CP/M BBS in the local area. I was having some difficulties, but was unaware that I was simultaneously giving the SYSOP a bad time!
Pretty soon, I received a somewhat indignant voice phone call from the SYSOP, wanting to know what the #$%^&* I was up to in my efforts to crash his BBS? I pleaded innocence, and he was a reasonable man, so we cooperated. He was handicapped because he was not a programmer; a third party was maintaining the software. Anyway, the problem was tracked down to the fact I was using the Model I backspace character 08h to correct my typing errors. This was OK of itself, but at his end, it meant I was backing up in his incoming text buffer to correct errors. In my own ignorance, I would occasionally backspace to the beginning of a line, and then hit a few more backspaces for good measure. This had no effect on my terminal program, but it had a dramatic

effect on his BBS software. It seems there was no protection to keep a user from backspacing out of his buffer into the prior code, and my backspacing would clear his buffer, and then begin to clear his operating code, which sooner or later crashed his BBS! And all done innocently! I was hacking without knowing it! The fellow maintaining the code then put a check feature into his code to prevent a repetition of what I had inadvertently done.

Another of my adventures goes back to the earliest times of the Model I. Some few of you may remember the monitor program RSM, which was quite powerful. You may also remember the infamous tape cassette system which worked some of the time. RSM had provisions for the user to add one additional command of his own creation. Some of the tapes of that time were copy protected by some kind of strange formatting, I naturally wanted to know how they worked. Since the tape initially had to load under Radio Shack's loading routine, I learned how that worked by disassembling the appropriate part of the ROM. I then created an 11 byte routine which became the User command added to RSM. This command would call the RS loader, and execute it to a point. It then proceeded to load all bytes from the tape thereafter into memory, and return control to me. I then studied and disassembled the code found on the tape, and quickly discovered that some of the "protected" tape programs simply used the RS loader to load a special loader written by the tape program author; once in place, this special loader code took over the loading of the remainder of the program and would cover its own tracks in various ways, one of which, I discovered, was to erase itself after execution. Cute! Anyway, I learned a lot about Z-80 code and a little about hacking from the good old RSM monitor. Incidentally, that monitor was originally written for CP/M, and was simply rewritten by its publisher to operate in the Model I environment. It also included a simple disassembler, which I used to disassemble itself! One must be imaginative, don't you know?

Roy Soltoff has written and sold a good disassembler, DSMBLR, which with some careful attention by the user, can produce good disassemblies of Z-80 code. Since one problem in disassembly is distinguishing between executable code and text and/or graphics, the operator must get closely involved in the process to make sense out of strange code. Roy's disassembler allows you to guide it and tell it which code is code to be disassembled, and which code is just ASCII text to be compressed into sentences. Roy's code also creates a reference table which summarizes all forward references. This is very helpful in disassembly analysis. A missing feature, however, is back references. That is, it is very helpful to know how the program got to point B; that is, where is point A that contains a call or other reference to point B?

There is also a disassembler, DISASSEM, built into NEWDOS80, Version 2, one of the more powerful alternate DOSes available on the Models I and III. This other, less powerful disassembler will, however, create a back reference table which, when used with Roy Soltoff's disassembly, greatly facilitates understanding of strange code.

Roy Soltoff writes sometimes mysterious code for the purpose of compactness, and I have had to puzzle over the result more than once. One of his tricks was to write a whole series of LD(IX+d) instructions, each with a different value for d, the index. If you began executing at the beginning of any of these instructions, you loaded the IX register several times, and then ignored it. What was he up to? Careful reading of the code showed he was entering, not at the beginning of an instruction, but in the MIDDLE of an instruction, effectively turning a 3 byte instruction into some other 2 byte instruction. Why? the result was that he could enter a block of code at two or more different points, set an index value in register C, and then continue on in the sequence with the desired index value in the C register, even after executing several other LD(IX+d) loads. This eliminated the need for a GOTO after each entry point. Roy is clever!

Some code tricks I have run into may be of interest to users. At one time, I spent a lot of time analyzing Super Utility to learn how it was protected. I did not analyze every version, but I will cite a couple of tricks by Kim Watt, the very clever author of Super Utility. He learned early on, that if you asked TRSDOS to copy a file, and if it discovered the checksum on a sector header or a sector of data was missing or incorrect, TRSDOS would simply supply the missing checksum AND SAY NOTHING about it. Kim used this feature in a clever way. Since the floppy disk controller actually will report this as an error, Kim would deliberately create a bad sector somewhere in his code by omitting the instruction to write a checksum. When his protected code was loaded and executed, it would examine the particular sector that should have an error in it. If the error was present, Super Utility knew the code had been loaded from the original disk and would proceed to operate. But if the disk was a bootleg copy, Super Utility would NOT find the error in the sector where Kim had placed it, and the code then knew it was from a copied disk, and would bomb out with an error message. Cute, eh? Another of his tricks I found on one of his Model I versions was a track with only two sectors on it, and these were 17h and 73h, both of

which were way above the normal sector numbers expected by TRSDOS. The DOS copy function did not know what to do with these two sectors, and ignored them. If you tried to run Super Utility from this version, it would examine that special track for sectors 17h and 73h; if not present, Super Utility knew it was a copy and would halt. The numbers 17 and 73 taken together are 1773, which was the type number of the floppy disk controller chip in the Model I, if you remember.

Many users objected to the fact that Super Utility would only run on its own protected, original disk, and could not be made into a /CMD file. Originally, Kim would sell a backup copy for an extra fee, $5, if I remember. Later, he made a /CMD version available for extra dollars, which met the desires of those who objected to having to boot a special disk.

While Super Utility was not written to do so, I discovered a way to use it to read the second side of Montezuma Micro CP/M disks, which extended its utility. Of course, most of Super U's functions would not work on CP/M, but the simple ability to read the disks was very helpful. You simply had to tell Super U it was going to read DOUBLE SIDED Model III disks, (which didn't even exist), but the Super U code would then successfully handle double sided CP/M. That sure helped me a lot when I was exploring CP/M on the Model 4.

Super U will also read IBM floppies, but only the first half of each sector, since IBM uses 512 byte sectors, and Super U expects 256 bytes. Not much value to it, but it works.

Thinking of other things I have found, I once disassembled a diagnostic program in which the author EXECUTED his Copyright notice! Since the notice was all ASCII, the characters, taken as executable code, consisted mostly of a bunch of register Loads, which accomplished nothing in the CPU. After completing execution of the copyright notice, the code continued on in a more customary fashion, but it was momentarily confusing to me as I labored to understand what he did. This was the same program which, after loading itself, went back and erased its own special loader so that if you interrupted it with the break key and executed DEBUG, the special loader was nowhere to be found.....

This same program had a branch where it asked the user if he wanted to test the memory or a disk drive? After some little fooling around, the program would load the L register, and immediately JP (HL). What the heck was in H? I had to go back half a page of code to discover where he had loaded H register to determine where JP (HL) was going to take me.

Self-modifying code is always described as a TERRIBLE thing, and I agree it can sure lead to trouble in executing a program. But it also provides some cute tricks in hiding code! Overwriting a no longer needed loader with later code is one example of this. If you try to break the program and go in and look around, part of the code is no longer there.

I remember a tape program which put a copyright notice on the screen. At that time, many bootleggers would go in and look for the copyright notice and delete it so they could pretend innocence about having a bootleg copy of someone's copyrighted program. In this particular program, the author had encyphered his copyright notice and something else, I forget what the other part was. He used two different encyphering methods. In one case, he had added a constant without carry to every byte of his copyright, which created bytes in the upper 128 characters of the ASCII series. To display the notice, he then subtracted without carry the same value, and put the resulting ASCII values up on the screen. In the second block of text, he used a different trick. There, he used a rotate instruction RRC to shift all the bits of each byte by one position. At display time, he would rotate the bits in the opposite direction with the RLC instruction and then put the resulting ASCII text on the screen.

How about a Radio Shack Hack? Old timers know how unreliable the original cassette tape loading scheme was. It required careful adjustment of the playback volume control to find a setting where the tape would load reliably. All kinds of tricks, both hardware and software were tried by many people in an attempt to get the darn thing to work properly.

The sad part of that story is that RS themselves made a mistake in coding the Model I ROM. That ROM, in fact, went through several revisions in its lifetime, but it wasn't until nearly the end of the Model I that RS found and corrected the error. When they did discover the fix, they opted not to correct the earlier ROMs in the proper way, which would have been to issue new ROMs. Cost, of course, was the reason. They also did not publicly announce the problem or its solution. What they did was to come up with a small board with two chips on it, called the XRX fix. IF you learned about its existence, you could turn in your Model I keyboard and they would install the XRX. They might or might not charge you for it. (They charged me). But the results were miraculous. My machine went from having a useable volume control range of about 1/2 number to a range from #1 to #10 on the wheel, completely reliable. After that I just set it at 5 and forgot about it. It always worked after that. The chip fix overcame a software mistake in a timing loop, and that was it.

Just imagine how much better the Model I would have been if that ROM had been correct originally! The tape system was well-designed, but the ROM mistake earned it a terrible reputation.

How many of you remember and UNDERSTAND PDRIVE as used in NEWDOS80? The PDRIVE command was one of the things which gave NEWDOS its great flexibility, but was simultaneously the bane of many users use of NEWDOS. It was entirely possible to format a disk in an unusual configuration, forget to write the PDRIVE on the outside of the jacket, and then find yourself unable to access the disk ever again. I know, I did it!

To resolve this, I did a little hacking. I found that NEWDOS80 creates a PDRIVE table on track 0, sector 1 in which all the PDRIVE information is neatly recorded. Of course, a lot of it was bit-mapped, which made understanding a bit difficult. I sorted this out by simply changing one parameter at a time, examining the sector 1 table after each change to see what changed. I screen-printed them for permanence. With this data in hand, it wasn't difficult to discover and record what each byte and bit accomplished. With this data in my notebook, I could easily reconstruct the correct PDRIVE line for any strange NEWDOS80 disk. Just another bit of hacking.

In conclusion, I can only regret the corruption of the hacker label; Life used to be more fun, now we are tagged (by the news media) as bad guys. What can we call ourselves that reflects the old-time hacker ethic, and which will not simultaneously bring down disapproval upon us? All suggestions welcomed.

I enjoyed exploring my personal memory for items to put into this article; I hope you enjoyed reading it.