TRSTimes

Volume 8. No. 1 - Jan/Feb 1995 - $4.00

OUR 8TH YEAR WITH THE TRS-80

LITTLE ORPHAN EIGHTY

I am so glad to have completed this issue of TRSTimes, because this is one that almost never was...

The Sunday after issue 7.6 was finished and in the mail, I just plain passed out. I had not slept for 3 days due to what I thought were ulcer pains. My wife, Sylvia, and son, Steven, managed to drag me down to the garage and into the car, and they drove me to the Westlake Community Hospital. There, after several tests, it was determined that I needed a gall-bladder operation. I didn't care what was wrong with me, I just wanted to sleep!

Supposedly, the operation was no big deal. The best-case scenario was that they might be able to use a new technique to pluck out one or more offending gall-stones - and I would be good as new in a day or so. The worst-case scenario was that they'd have to use the old procedure and remove the gall-bladder completely. Again, no big deal - my wife was told that I'd be fine in a couple of days. They doped me up and wheeled me to wherever it is they cut people open. I was out like a light before we reached the door.

It turned out to be somewhat more serious than anticipated. My gall-bladder needed to be completely removed, and then complications set in - the poison in my body spilled into my lungs, and I ended up with double pneumonia.

When my wife next saw me I was not in the expected recovery room, but in the Intensive Care Unit, fighting for my life. I had tubes down my throat, tubes up my nose, tubes in my arms and my hands were tied to the sides of the bed so I wouldn't rip out those intrusive objects. I was hooked up to a respirator - I must have been a pathetic sight. One slight consolation was that I had a button to push if I felt pain. *Good stuff - legalize morphine!!*

I was in ICU for 5 days, and then spent another 5 days in a regular hospital room. I have never been sick before in my life, so this was, to say the least, an interesting experience. The highlight of my stay was my first meal - a breakfast of some kind of broth - man, was that good!

10 days in the hospital is a long time and, after I had accustomized myself to my plight, I tried to do what I always do when I am away from my computer and bored - I tried to write a program in my head. *But I couldn't do it!!*

That startled me. Why couldn't I do now what had been so easy for me just a few days ago? The answer, of course, was that I was pumping morphine into my body, causing my brain to slow down considerably.

This condition lasted even as I left the hospital and took up residence in my own bed. My Model 4P was sitting on a cart immediately next to me, and I would turn it on at various times of the day and night, hoping that I could do something productive. At first I struck out hopelessly, but eventually the fog lifted; luckily, just in time to get down to work on this issue of TRSTimes — and finish it on schedule.

I am very appreciative of the help and support I have received from my friends in the TRS-80 community. It certainly made putting out this issue much easier than I had anticipated.

Frank Slinkman sent a super installment of his C tutorial. I do hope that our readers are taking advantage of Frank's knowledge and get started learning this powerful language.

Danny Myers provides all the answers to two of the famous Scott Adams adventures. I am glad that so many of you are now playing those games again. To the ones expressing concern that the walkthroughs diminishes the playing enjoyment, let me say that I'm sure that it was never Mr. Myers' intention that you should use the articles as a playing bible; rather, you should only use his writings if you get REALLY stuck.

Chris Fara explains the mysteries of integers, single- and double precision numbers. After reading this article, you might just be qualified to help Intel fix the problem with the Pentium chip.

These days there are a lot of Deskjets attached to TRS-80. They are good printers and Doug Hyman tells us how to save money by refilling those expensive ink cartridges.

Kelly Bates shows how to use NEWDOS/80 to transfer TRSDOS 1.3 files.

James Sowards presents a fun card trick for Model 4. Try this one — you'll like it.

Roy Beck comes up with yet another of his essays on difficult subjects. The mixture of reminiscences and facts make reading about the Z-80 instruction set fun. Imagine that!

Finally, the Odds & Ends piece is my baby. I hope that my foggy brain managed to produce something readable.

And now.....

Welcome to the 8th year of TRSTimes.

TRSTimes magazine

Volume 8. No. 1 - Jan/Feb 1995 - $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.

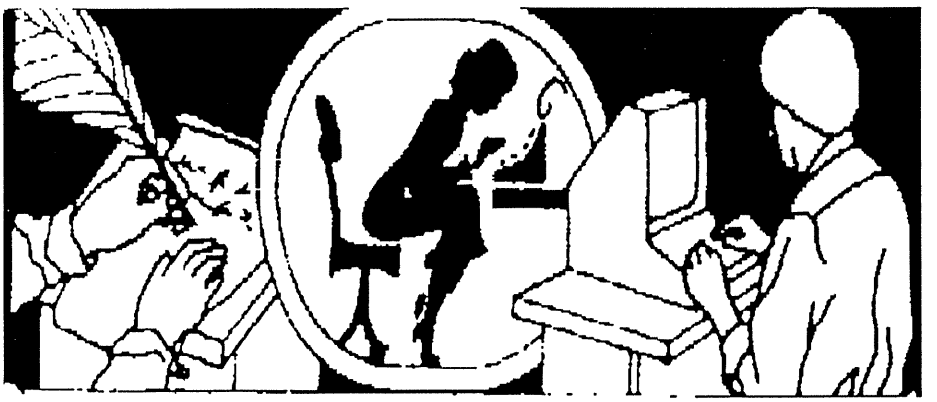
Publication months are January, March, May, July, September and November.

1995 subscription rates (6 issues):
UNITED STATES $21.00
CANADA $22.00 (U.S.)

EUROPE, CENTRAL & SOUTH AMERICA:
$26.00 for surface mail or $34.00 for air mail. (U.S. currency only)

ASIA, AUSTRALIA & NEW ZEALAND:
$28.00 for surface mail or $36.00 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.



ODDS & ENDS FOR MODEL 4

TRSDOS/LS-DOS 6.3.x.

by Lance Wolstrup

When TRSTimes was first published, I began a column called HUNTING FOR BURIED TREASURE. The purpose was to dispute the 'experts' who, along with the Model 4 manual, claimed that there was no longer any reason to POKE values to strategic DOS memory locations as had been the norm for BASIC programmers on both the Model I and III.

I quote from issue 1.1:

The author of TRSDOS/LS-DOS 6.x.x, Roy Soltoff, was very careful to isolate us from the actual addresses of important data areas and DOS routines. Instead, Assembly Language programmers were given SuperVisor Calls, and the BASIC programmer was given the clumsy SYSTEM command. Mr. Soltoff explains the reason for this:

"Trying to keep the memory locations data constant across all implementations of the system is quite restrictive and usually becomes limiting to the healthy growth of the system. Keeping portability in mind, the designers of the system have provided SuperVisor Calls which return pointers to data that may be useful to a program. Thus, there should usually be no need to access data areas by memory address."

I went on to protest that since there would, most likely, not be any growth, healthy or otherwise, we need not concern ourselves with changing data locations; rather, we were free to explore and use what we found.

As it turned out, I was wrong. There was to be one more DOS upgrade - from LS-DOS 6.3 to LS-DOS 6.3.1. However, the vast majority of the goodies we unearthed were still valid in the new version. Only a few programs, that for one reason or another needed to break the rules, failed to work. The most famous of these is probably the BOOT5/CMD program which was fixed right here in the pages of TRSTimes.

We proceeded to publish several hidden goodies, some were straight forward, while some were of the rather exotic variety, and I have written numerous programs employing the techniques and memory locations described in our pages.

A good example is to imagine a program that asks the user if he/she wishes to play again. That would written something like this:

```
800 print"Would you like to play again (Y/N)";
810 i$=inkey$:if inkey$="" then 810
820 if i$="Y" or I$="y" then 100 'play again
830 if i$="N" or i$="n" then cls:end 'don't play again
840 goto 810
```

Notice that the program needs to check either response for both upper- and lowercase input. Well, in this case, that is not a big deal.

However, now imagine that your program needs to match the user input against an internal table, such as, for example, a list of capitol cities. The user might be asked:

What is the capitol of Ohio?

The correct answer residing in the internal list is COLUMBUS, so if the user types Columbus, COLUMBUs, or any other combination of upper- and lowercase spelling, the answer will be deemed wrong.

The obvious fix is to convert the entire user input to uppercase, and then make the comparison against the table. This would usually be done something like this:

```
200 print"What is the capitol of OHIO?";
210 input i$
220 for x=1 to len(i$)
230 if mid$(i$,x,1)=>"a" and mid$(i$,x,1)<="z" then
i=asc(mid$(i$,x,1)) and 223:mid$(i$,x,1)=chr$(i)
240 next
250 if i$="COLUMBUS" then print"Correct" else
print"Wrong"
```

It is plain to see that the code in lines 220 through 240 is lengthy and difficult. It works, but there certainly is a better way - and HUNTING FOR BURIED TREASURE published it in issue 1.1:

POKE &H74,PEEK(&H74) OR 32

This sets bit 5 of memory location 74 hex (known as KFLAG$). Bit 5 of this location controls the keyboard case setting. On (1) is uppercase, and Off (0) is lowercase.

The problem with this POKE is that, while it does force the keyboard into uppercase, it is possible for the user to press <SHIFT><0>, changing the case setting back to lowercase. I got around this limitation by not using the INPUT command, but instead route all user input through a special INKEY$ routine where each keystroke was forced to uppercase. However, if the user really tried to defeat the uppercase conversion, even this routine could fail by leaving the very first character in lowercase.

It was time to solve this annoyance once and for all, so I began reading THE SOURCE (Roy Soltoff's commented disassembly of the entire TRSDOS 6). While it is not your typical light reading, it is informative and, sure enough, before long the answer to the problem appeared on the page.

The keyboard driver tests if the caps-lock key is active and, if it is, it jumps to a routine that then convert the character to uppercase. The code goes like this:

```
0A09 CB6E   BIT   5,(HL)          ;caps lock?
0A0B 2031   JR    NZ,TGLCASE ;jump if yes
```

Right there at memory location 0A0B hex we have the solution. All we need to do is to change the byte from the conditional JR NZ to a simple, unconditional JR; that is, change:

```
0A0B 2031   JR    NZ,TGLCASE
```

to

```
0A0B 1831   JR    TGLCASE
```

In other words — we only need to change the byte at 0A0B hex from 20 hex to 18 hex.

POKE &HA0B,&H18

Voila, we now have constant uppercase. This setting will remain intact until the system is reset by a reboot, or you poke back the original value:

POKE &HA0B,&H20

This makes it much easier for a program to check user input — just wish I'd thought of it sooner.

Next on the agenda is an item that falls in the "USELESS - BUT FUN" category.

The Model 4 is blessed with two library commands that will display the names of the diskfiles on the screen: DIR and CAT.

The DIR command displays a complete directory listing, while the CAT command list 5 filenames per line with no extra information. At first glance the commands appear to be vastly different — but are they?

The answer is NO. They are basically the same command.

CAT (A=Y)
is the same as typing DIR

and

DIR (A=N)
is the same as typing CAT

Now you know!

A LITTLE BOOLEAN

I have often needed to determine whether a number input by the user was odd or even. This is a simple programming task, and I have usually written the code as follows:

```
100 PRINT"Type a number ";
110 INPUT N
120 IF N/2<>INT(N/2) THEN PRINT"ODD" ELSE
PRINT"EVEN"
```

Line 120 performs division before the comparison determines whether the number is odd or even. A much more elegant way to accomplish this task is to use Boolean logic. Change line 120 to:

```
120 IF N AND 1 THEN PRINT"ODD" ELSE
PRINT"EVEN"
```

MORE BOOLEAN

Did you know that ...

the + sign can be used instead of OR

```
10 IF A<32 OR A>122 THEN....
```

can be written

```
10 IF (A<32)+(A>122) THEN.....
```

the * sign can be used instead of AND

```
10 IF A=2 AND B=3 THEN ...
```

can be written

```
10 IF (A=2)*(B=3) THEN...
```

the - sign can be used instead of XOR

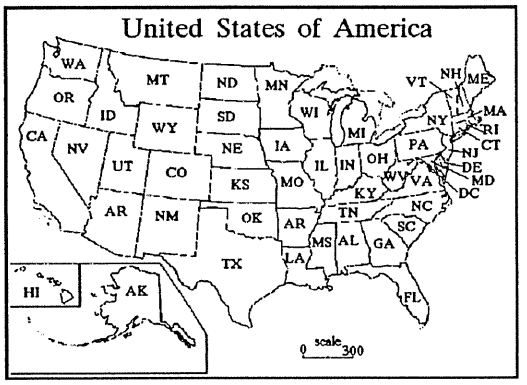
```
10 IF A=2 XOR B=3 THEN PRINT"TRUE" ELSE
PRINT"FALSE"
```

can be written

```
10 IF (A=2)-(B=3) THEN PRINT"TRUE" ELSE
PRINT"FALSE"
```

Test this one by making line 5, first A=2; then B=3, and finally A=2:B=3.

A GEOGRAPHY GAME



Recently I borrowed a stack of IBM shareware disks from schoolteacher Ann Collins, friend, VTUG member and TRSTimes subscriber. Among the programs I looked through was one written in GW-Basic, an educational geography game about U.S. states and their capitols. This interested me, because I had written a similar one for my son, Steven, which was published in TRSTimes some years ago.

The IBM program was not particularly well written, but it did have a couple of interesting features. It offered two games in one — What is the capitol of... and What state has the capitol city... — and it had a hint module. Boy, I wish I'd thought of those when I wrote mine!

But since it is never too late to correct one's omissions, and since I needed a program to demonstrate the tricks from this article, I stole the program. Well, I didn't really steal it, and I especially did not steal the code; rather, I played with the PC program a few times and then rewrote it from scratch on my Model 4.

The program, now called States and Capitols, opens with a menu of 3 choices. Option 1 plays the "What is the capitol of" game. Option 2 plays the "What state has the capitol city of" game. Option 3 exits the program back to Basic. Both the games ask a series of 20 questions and a running score is kept at the bottom of the screen. If the user gives an incorrect answer, a hint is offered. The hint displays a random letter in the answer.

Because I use the POKE to force the constant uppercase, I also use the POKE that disables the break key. This will force the user to exit through my exit routine where all the POKEs are reset to normal. Also note that most prompts accepts the ESC sequence <Shift><Down Arrow> to move back one level, or from the first menu, exit to Basic.

The game is easy to play, and it is, indeed, a great teacher. Whether you are of school-age or, like my wife and I, on the road to 'senior-citizenship', STATES AND CAPITOLS can help you recall the names of those cities far away. Type in the listing below and have fun.

STATES AND CAPITOLS

```
1 'states and capitals
2 'for TRS-80 Model 4
3 'copyright 1994 by Lance Wolstrup
4 'all rights reserved
5 '
10 DEFINT A-Z:SW=80

                    poke to force uppercase
11 POKE &HA0B,&H18

                    poke to disable break key
12 POKE &H7C,PEEK(&H7C) OR 16

15 DIM Q$(50,2),QF(50)
19 GOTO 100

                    universal print routine
20 H=0:GOTO 23
21 H=(SW-LEN(A$))/2:GOTO 23
22 H=SW-LEN(A$)
23 PRINT@(V,H),A$;:RETURN
```

universal input routine

```
30 FL=0:L=0:I$="":A$=STRING$(ML,46):GOSUB 23:
A$=CHR$(14):GOSUB 23
31 A$=INKEY$:IF A$="" THEN 31
32 IF A$=CHR$(13) THEN PRINT CHR$(15);:RETURN
33 IF A$=CHR$(27) THEN FL=1:PRINT CHR$(15);:
RETURN
34 IF (A$=CHR$(8))*(L=0) THEN 31
35 IF A$=CHR$(8) THEN L=L-1:I$=LEFT$(I$,L):H=H-1:
A$=CHR$(46):GOSUB 23:A$="":GOSUB 23:GOTO 31
36 IF (A$<CHR$(32))+(A$>CHR$(122)) THEN 31
37 IF L=ML THEN 31
38 GOSUB 23:H=H+1:L=L+1:I$=I$+A$:GOTO 31
```

hint routine

```
40 V=12:A$=CHR$(30)+"Incorrect answer -- would you
like a hint (Y/N)? ":GOSUB 21:H=H+LEN(A$)
41 ML=1:GOSUB 30:IF I$="Y" THEN 42 ELSE IF
I$="N" THEN F=1:H=0:A$=CHR$(30):GOTO 20 ELSE
40
42 R1=RND(LEN(AN$)):IF MID$(HI$,R1,1)<>CHR$(46)
THEN 42
43 MID$(HI$,R1,1)=MID$(AN$,R1,1)
44 V=12:A$=CHR$(30):GOSUB 20:
A$="Here is your hint: "+HI$:GOSUB 21:
H=H+LEN(A$):HI=HI+1:IF HI$<>AN$ THEN 47
45 V=15:A$="Press <ENTER> to continue ":GOSUB 21:
H=H+LEN(A$)
46 ML=1:GOSUB 30:IF I$="" THEN A$=CHR$(30):
GOSUB 20:F=1:V=12:GOTO 20 ELSE 45
47 RETURN
```

program begins here

```
100 PRINT CHR$(15):CLS
110 V=1:A$="S T A T E S  a n d  C A P I T O L S":
GOSUB 21
120 V=2:A$="an educational game for TRS-80 Model 4":
GOSUB 21
130 V=3:A$="copyright (c) 1994  by Lance Wolstrup":
GOSUB 21
140 V=4:A$=STRING$(SW,131):GOSUB 20
150 FOR X=1 TO 50:FOR Y=1 TO 2:READ Q$(X,Y):
NEXT:NEXT
```

display menu

```
200 V=9:A$=CHR$(31):GOSUB 20:H=20:
A$="1. What is the capitol of . . .":GOSUB 23
210 V=11:A$="2. What state has the capitol city . . .":
GOSUB 23
220 V=13:A$="3. Exit program":GOSUB 23:H1=H
230 H=H1:V=16:
A$="Please make your selection ( 1 - 3 ) ":GOSUB 23
240 H=H+LEN(A$)
250 ML=1:GOSUB 30:IF FL THEN 280
ELSE IF I$="" THEN 250
260 I=VAL(I$):IF (I<1)+(I>3) THEN 230
270 ON I GOTO 300,400,280
```

exit routine

```
280 POKE &HA0B,&H20:
POKE &H7C,PEEK(&H7C) AND 239:
PRINT CHR$(14);:CLS:END
```

select "What is the capitol of"

```
300 Q=1:A=2:GOTO 500
```

select "What state has the capitol city

```
400 Q=2:A=1
```

initialize for new game

```
500 RANDOM:QA=0:QC=0:HQ=18:HC=77:HI=0
510 '
520 FOR X=1 TO 50:QF(X)=0:NEXT
```

display scoreboard

```
522 V=6:A$=CHR$(31):GOSUB 20
525 V=22:A$=" Questions asked:":GOSUB 20:
PRINT@(V,HQ),USING"##";QA:H=60:
A$="Correct answers:":GOSUB 23:
PRINT@(V,HC),USING"##";QC:V=21:
A$=STRING$(SW,131):GOSUB 20
```

main gameloop

```
530 FOR X=1 TO 20
535 QA=QA+1
540 R=RND(50):IF QF(R) THEN 540 ELSE QF(R)=1
545 AN$=Q$(R,A):HI$=STRING$(LEN(AN$),46)
550 IF Q=1 THEN A$=CHR$(30)+"What is the capitol of
"+Q$(R,Q)+"? ": ELSE A$=CHR$(30)+Q$(R,Q)+" is the
capitol of which state? "
560 V=9:GOSUB 20:H=H+LEN(A$)
570 ML=LEN(AN$):GOSUB 30:IF FL THEN X=50:
GOTO 600
580 F=0:IF (I$="")+(I$<>AN$) THEN GOSUB 40:
IF F THEN 595 ELSE 550
590 QC=QC+1:V=12:H=0:A$=CHR$(30):GOSUB 20
595 V=22:PRINT@(V,HQ),USING"##";QA:
PRINT@(V,HC),USING"##";QC
600 NEXT:IF FL GOTO 200
```

display game result

```
610 V=12:A$="You correctly answered"+STR$(QC)+" out
of"+STR$(QA)+" questions":GOSUB 21
620 PC=(QC/QA)*100
630 V=14:A$="for a score of"+STR$(PC)+"%":GOSUB 21
635 V=16:A$="You used"+STR$(HI)+" hints":GOSUB 21
```

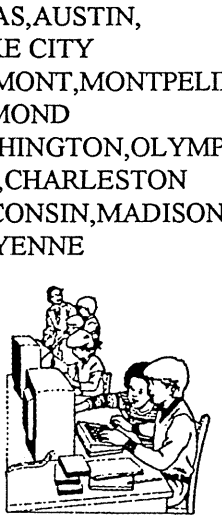
play again routine

```
640 V=19:A$="Would you like to play again (Y/N)? ":
GOSUB 21:H=H+LEN(A$)
650 ML=1:GOSUB 30:IF I$="" THEN 650 ELSE IF
I$="N" THEN 280 ELSE IF I$="Y" THEN 200 ELSE 650
```

data statements

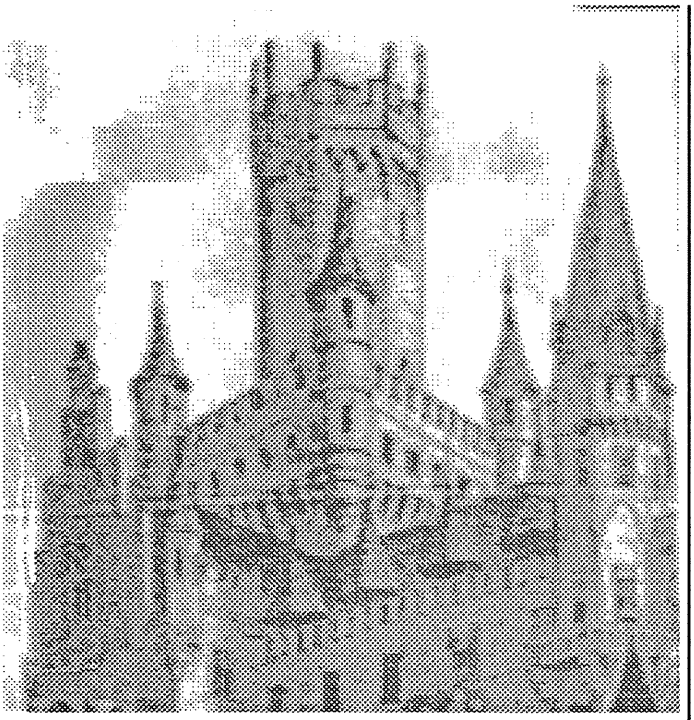
```
1000 DATA ALABAMA,MONTGOMERY,
```

```
ALASKA,JUNEAU
1010 DATA ARIZONA,PHOENIX,
ARKANSAS,LITTLE ROCK
1020 DATA CALIFORNIA,SACRAMENTO,
COLORADO,DENVER
1030 DATA CONNECTICUT,HARTFORD,
DELAWARE,DOVER
1040 DATA FLORIDA,TALLAHASSEE,
GEORGIA,ATLANTA
1050 DATA HAWAII,HONOLULU,
IDAHO,BOISE
1060 DATA ILLINOIS,SPRINGFIELD,
INDIANA,INDIANAPOLIS
1070 DATA IOWA,DES MOINES,
KANSAS,TOPEKA
1080 DATA KENTUCKY,FRANKFORT,
LOUISIANA,BATON ROUGE
1090 DATA MAINE,AUGUSTA,
MARYLAND,ANNAPOLIS
1100 DATA MASSACHUSSETTS,BOSTON,
MICHIGAN,LANSING
1110 DATA MINNESOTA,SAINT PAUL,
MISSISSIPPI,JACKSON
1120 DATA MISSOURI,JEFFERSON CITY,
MONTANA,HELENA
1130 DATA NEBRASKA,LINCOLN,
NEVADA,CARSON CITY
1140 DATA NEW HAMPSHIRE,CONCORD,
NEW JERSEY,TRENTON
1150 DATA NEW MEXICO,SANTA FE,
NEW YORK,ALBANY
1160 DATA NORTH CAROLINA,
RALEIGH,NORTH DAKOTA,BISMARCK
1170 DATA OHIO,COLUMBUS,
OKLAHOMA,OKLAHOMA CITY
1180 DATA OREGON,SALEM,
PENNSYLVANIA,HARRISBURG
1190 DATA RHODE ISLAND,PROVIDENCE,
SOUTH CAROLINA,COLUMBIA
1200 DATA SOUTH DAKOTA,PIERRE,
TENNESSEE,NASHVILLE
1210 DATA TEXAS,AUSTIN,
UTAH,SALT LAKE CITY
1220 DATA VERMONT,MONTPELIER,
VIRGINIA,RICHMOND
1230 DATA WASHINGTON,OLYMPIA,
WEST VIRGINIA,CHARLESTON
1240 DATA WISCONSIN,MADISON,
WYOMING,CHEYENNE
```



BEAT THE GAME

By Daniel Myers



THE COUNT

The Scott Adams Adventures

In this adventure, your goal is to find and kill the evil vampire, Count Dracula. This is probably the most boring of the Adams adventures because you have to do a lot of waiting for some things to happen, and exactly when they happen seems to be a random thing.

You start off in bed in Dracula's house. Get the sheets, then get up. Go North into the hallway, then West into the kitchen. Enter the dumb waiter and raise it, then go room. You are now in the pantry. Get the matches and the garlic, then return to the dumb waiter.

At this point, you can do some exploring to waste time, because you're waiting for the bell to ring (do a little mapping!). This will tell you that the postman has delivered some mail to the house. Once you hear the bell, go to the front door. You will find a postcard with a note clipped to it. Get the paper clip and drop the note.

Now, return to the kitchen, get into the dumb waiter, and lower it to the Work Room. Go into the room. Pick the lock on the closet door (with the paper clip, of course), open the door, and enter the closet. Drop the stake you're carrying, then get the vial and leave.

Back in the Work Room, close and re-lock the door, and drop the paper clip. Now it's time to get a light source, so go Down to the dungeon. Tie the sheets to the iron ring, then descend into the pit. Get the torch (it's there even though you can't see it), and climb out again. You don't have to bother with the sheets again for now.

Now go back up to the kitchen. Empty the vial (3 no-doz tablets come out) then drop it. Now you have to wait until sunset, but you must be careful here. Stop saying "wait" as soon as it starts to get dark outside, and take a tablet. Then continue to wait until nightfall, at which time, light your torch.

Wait a little longer, then enter the oven. It's a solar oven, so you can never get in here during the day. Get the nailfile that's inside, and leave the oven. At this point, you have done all you can, so you might as well go back to bed. Remember to unlight the torch before going to sleep. This night you will be bitten, and there is nothing you can do about it.

The next morning, you will notice that the sheets are on the bed again. Get them, then get up. Tie the sheet to the bed, then get the sheet (loose end), open the window, go out the window, and drop the end of the sheet over the ledge.

Now climb down the sheet, go to the Window Box, then go through the window into the room. Get the portrait of Dracula, then drop it. This will reveal a secret passage. If you want to explore it, go ahead (light torch first). Just make sure you unlight the torch before trying to climb back up the sheets!

Well, here you have to do some more waiting, as now you have plenty of time to kill until the mailman brings a certain package. So, just mess around again until the bell rings. At that point, go to the front door again.

Open the package. Inside are a bottle of blood and a pack of cigarettes. Get the cigarettes (make sure you say "pack"!). Now go back to the kitchen and get the tablets. After that, go back down the sheets to the room with the secret passage. You

should do this before night falls.

Now wait for sunset the same way you did the night before. Once the sun sets, light your torch and go into the passage. Follow it along to the crypt. Get a cigarette (you may have to drop something first), then smoke it. Dracula's (empty) coffin will appear (this is the only way to make the coffin appear).

Open the coffin and go inside. Use the file to break the bolt on the coffin, then get out again. Drop the cigarette, pick up anything else you may have dropped, and leave the passage. Now, take a second cigarette from night, it really isn't necessary, since you can take three bites before succumbing. So, you might just as well put out the torch and go to sleep.

Once again, you wake up in the bed, with the sheets. Get them, and tie them to the bed as you did the day before. Instead of climbing down however, first go to the dumb waiter and from there to the Work Room. Use the clip to pick the the lock on the closet door so you can get the stake. Also make sure you grab the mallet on your way out.

Now that you have the stake and the hammer, go up to the kitchen again, and from there to the bedroom and down the sheets to the secret passage. Light the torch, enter the passage, and go to the crypt. Smoke the cigarette, and the coffin will appear again.

Open the coffin, go inside (yes, you must!), and apply the coup-de-grace to the vampire. TA-DA! You're a hero! (Hey, you're pretty good at this stuff.)

ADVENTURELAND

The Scott Adams Adventures

Here you stand in a forest, about to start your adventure. Move along East, and tiptoe past the sleeping dragon. Go East again to the lake. Get the axe (leave the fish for now, since you have no way of getting them yet), then head North into the quicksand bog. Pick up the blue ox (WOW! A treasure already!, then say the magic word on the axe. Gee, where did everything go? Well, look at it this way: at least you can "Swim" back to the lake.

Now go South to the bottomless hole. Carefully "Go Hole," and pick up the flint and steel. Go up (easy, it's a long way down!). Then it's West into the swamp, and West again to the hidden grove. Surprise! There's the axe and the ox, along with another treasure. Get everthing, and go back East. Climb the tree, get the keys, and climb down again. Now, "Chop Tree." Drop the axe (you won't need it again), get the mud, then, "Go Stump." Once inside the stump, drop the mud, ox and fruit. Go down to the root chamber, pick up the rubies, then go up and drop them off. Easy, isn't it?

Okay, time for more treasures. Get the lamp and rub it twice. Each time you rub the lamp, a genie appears and leaves a treasure. Just make sure you don't rub it more than two times, or the genie will come out and **TAKE AWAY** a treasure! (Can't win the game that way!) Now, go down into the root chamber, and "Go Hole." Open the door and drop the keys. Light the lamp (it's dark up ahead), and "Go Hall." Then continue down to the cavern. From the cavern, trek South and pick up the bladder. Now it's time for a return trip, so move along North, then up until you're out of the stump and back in the swamp again (by the way, any time the chiggers chomp on you, just get the mud, then drop it again. Also, "Unlight Lamp" when you get back to the root chamber...energy conservation helps!). "Get Gas," then "Go Stump."

Now head back down to the cavern (remember to light the lamp before going into the hallway!), then go South and up. Drop the bladder and "Ignite Gas." *BOOOOM!* You just blew a hole in the bricked-up window. "Go Hole," then jump (don't worry, you can make it to the other side safely). Ummmm, hello, bear! Well, he doesn't look too mean, but it's better not to take any chances, so "Yell." As the startled bear falls down towards the bottom of the chasm, pick up the magic mirror (carefully, it's **VERY** fragile!), then "Go Throne." Grab the crown and go West to the ledge. Jump over again, then West. Pick up the fire bricks on your way out (heavy stuff, but you'll be needing them soon), and go down and North. After that, make your way up again to the treasure room in the stump.

Drop off **ONLY** the crown. **DO NOT** drop the mirror! Okay, now get the bottle of water, and go down again. This time, drop the flint just before you go into the hallway. Go down to the cavern, and from there, down again into the maze of pits. From the "Opposite of Light" sign, go down, West, and down. Pick up the rug, then go down again and you're at the bottom of the chasm. "Build Dam," then drop the remaining bricks (you had a few left over). "Look lava," and there's a firestone. It's still pretty hot, so "Pour Water." Now get the stone and the golden net. Hmmmm, but how to get out of here? Well, that sign might help. Say Away" twice and, voila! you're in the meadow again (ahhh! fresh air!). Unlight the lamp, then go South (to the swamp) and over to the stump.

Drop the firestone, the rug, and the mirror. Make sure that you drop the rug **BEFORE** you drop the mirror! (Otherwise, seven years of bad luck and you won't be able to finish the game!) Now, get out of the stump, go East to

the hole, and North to the lake. "Get Water," then "Get Fish" (can't get them without the net, you see). Return to the stump, and drop off the fish and the net, in that order. Pick up the mud, and head back down to the cavern. Remember to get the flint and light the lamp before entering the hallway!

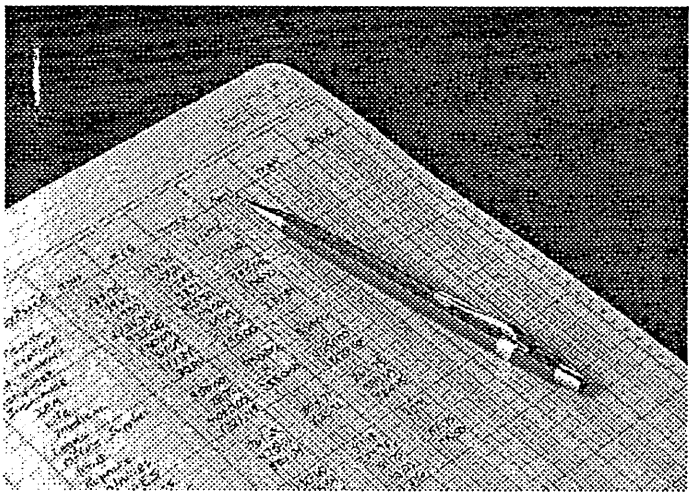
Once in the cavern, go North. At this point, you should save the game. Up ahead are the African bees, which you will need to get rid of the dragon. Unfortunately, this part of the game seems to be random, and sometimes the bees will suffocate and die before you can bring them out. I have never found a surefire method for keeping the bees alive, so save the game here and hope you won't have to restore it too many times! Okay, now that the game is saved, go North again. Get the honey, pour out the water, and get the bees. (If the bees sting you, and you find yourself in limbo, either restore your saved game or "Go Up" to return to life in the outdoors.)

Once you have the bees, head back South to the cavern, then all the way back up to the swamp. If the mud hasn't fallen off yet, drop it here before going on (the dragon **HATES** the smell of mud, and will most certainly kill you if you go near her with it -- yes, **HER!**>. All right, now head North and "Drop Bees." The dragon will become annoyed and fly away, leaving behind some precious and rare eggs. Pick up the eggs, then return to the stump. Drop the eggs and the honey, then say "Score."

***** YIPPPPEEEE! *****

You did it!! (Whew! You deserve to take some time out now and relax! But wait...could that be a pirate flag I see on the horizon...?)

PROGRAMMING TIDBITS



Some numbers have a point

Integers are by far the most efficient numbers in computers. Integer math is fast, memory storage requirements are minimal, encoding and decoding are quite straightforward. Integers are the "natural" species of numbers in the binary world of computers. Practically every programming text emphasizes that integers should be used whenever possible. The rub is in the "whenever" because in many computations fractions cannot be avoided. That's where the "floating point" numbers come in. In our BASIC's they are known as "single precision" and "double precision" numbers. In other programming languages they may be called "real numbers", "floats", "doubles", etc.

The basic principle of binary numbers with fractional parts is similar to the decimal system. In the decimal system a point divides the number in two parts. To the left of the point the positions of digits represent increasing positive powers of ten. To the right of the point the positions of digits represent decreasing negative powers of ten. A negative power is the same as one divided by the positive power. The first position to the right of the point represents 10 to the power of -1 or 1/10, the second position 10 to -2 or 1/100 and so on. The value of the fraction is the sum of those powers, each multiplied by the corresponding digit. For example in decimal 0.23 it is 2*(1/10)+3*(1/100) or 23/100.

Same idea holds in binary, except that the positions of bits to the right of the "binary point" represent decreasing negative powers of two, instead of ten. The first bit to the right of the point represents 2 to the power of -1 or 1/2, the second 2 to -2 or 1/4, then 1/8, etc. In a way this is even easier than decimal fractions, because bits can be only "1" or "0". Thus the value of a fraction is simply the sum of those negative powers of 2 which have a "1" in their positions. For example the binary fraction.....

0.1011

has the value 1/2+0+1/8+1/16 or, after bringing it all to the common denominator, 11/16 or decimal 0.6875. The concept is somewhat similar to the traditional "English" measuring system where scales and tapes are calibrated in inches and fractions of inches, using ever smaller ticks for each next half of the previous fraction: 1/2, 1/4, 1/8, etc.

There wouldn't be any big mystery in all that, except that here again as with the plus or minus in "signed" integers, the computer has a handicap: it knows nothing about any "point". In most programming systems this problem has been solved by associating the bits of a fractional number with a separate "exponent" byte. To see how that works, let's assume a very primitive "floating point" system where all numbers are encoded in only one byte. In such a system we might have a number like this.....

101.10000

whose value is 4+0+1+1/2+0+0+0+0 or decimal 5.5. Now if we move the point one place to the left.....

10.110000

then the value is 2+0+1/2+1/4+0+0+0+0. As we can see, the value of this number is exactly 1/2 of the original value. If we now move the point two more places to the left, we get.....

.10110000

The value of this "normalized" number is now.....

1/2+0+1/8+1/16+0+0+0+0

or decimal 0.6875. In other words, each time we move the point to the left, we divide the number by 2. To get back the original number, we would multiply this "normalized" value by 2 as many times as the point was shifted. Now suppose we have a small fraction.....

.00001011

Its value is 1/32+0+1/128+1/256, decimal 0.04296875. To normalize it, we shift the point to the right until it is in front of the first "1". After four shifts we get the normal form (padded with zeros at the end to fit a whole byte).....

.10110000

whose value is 1/2+0+1/8+1/16, decimal 0.6875. To restore the original value, we would divide this value by 2 as many times as the point was shifted.

The "floating point" numbers are stored in such a normalized form called "mantissa", with the point always assumed to be in front of the first significant bit. The number of shifts is encoded in the "exponent" byte.

The exponent byte must differentiate between shifts to the left and right. Since a byte can hold an absolute (unsigned) value up to 255, "Microsoft" programming systems such as our TRS-80 BASIC, adopted a convention that 128 (halfway between 0 and 255) means no shifts. If the shifts were made to the left (i.e. the original value was 1.0 or greater) then they are encoded as 128+E, while shifts made to the right are encoded 128-E, where E is the number of shifts. In our first example we have made three shifts to the left, so the exponent would be 128+3 or 131. In the second example the four shifts to the right would produce the exponent 128-4 or 124. The exponent byte goes in front of the "mantissa". Thus in the computer's binary language our two examples might look like this (of course in the computer there is no "point"; we only write it here as a reminder that we are talking about "floats").....

10000011 .10110000	decimal 5.5
01111100 .10110000	decimal 0.04296875

Lo and behold, with this exponent trick two totally different values got boiled down to identical mantissas. The only difference is in the exponent. Other examples.....

10000000 .10110000	decimal 0.6875
10000100 .10110000	decimal 11.0
10001000 .10110000	decimal 176.0

In the first example the value of the exponent is 128, so there are no shifts and the mantissa bits are taken at their "face value" 1/2+0+1/8+1/16 or 0.6875. In the second example the exponent is 128+4, so 4 shifts restore the original binary number 1011.0000 which is decimal 11.0. The third exponent 128+8 shifts the point just beyond the last bit and the value of 10110000. gives 176.0. But what if the exponent shifts the point way past the last bit? For instance.....

10010000 .10110000	decimal ?

Now the exponent is 128+16, so to get the encoded value, we just assume that an infinite trail of 0's exists after the last bit of the mantissa, and shifting 16 times we get (only the expanded mantissa is shown below).....

10110000 00000000.000...	decimal 45056.0

Similary, if the exponent is less than 128 then we assume an infinite number of 0's in front of the mantissa and keep shifting the point as far back as necessary.

There is one special case in this system. When the exponent is "zero" then the value of the number is "zero", regardless of the mantissa bits. This is relevant because of the way this system deals with the "plus" or "minus" sign. Since by definition the first bit of the normalized mantissa must be always "1", we don't really need to keep this bit in the matissa. It is "redundant". Instead, we use this bit to represent the sign. The idea here is slightly different and actually simpler than in "signed integers" discussed last time (TRSTimes 7.6). The sign is plus when the first bit of the mantissa is "0", minus when that bit is "1", and that's it. Thus the final form of signed "floating point" numbers looks like this....

10000011 .00110000	decimal +5.5
10000011 .10110000	decimal -5.5

Unlike in "signed" integers, the only difference between positive and negative numbers is the first bit in the mantissa. To change a positive "float" to a negative, or the other way around, we only need to flip that one bit.

In summary, the encoding of floating point numbers in our TRS-80 computers is a 5-step process.....

[1] Express value as a sum of powers of 2.
[2] Write bit pattern, note position of the point.
[3] Shift the point left/right to normalize mantissa.
[4] Add/subtract shifts to/from 128 to get exponent.
[5] Put 0 or 1 into first bit of mantissa for plus/minus.

Decoding is the reverse.....

[1] Determine sign, then put "1" into first bit of mantissa.
[2] Subtract 128 from exponent to find how many shifts.
[3] Shift the point.
[4] Add up absolute values of bits in the mantissa.
[5] Make the total positive/negative depending on the sign.

Since the decoding process always sticks a "1" into the first bit of the mantissa, the result would be misleading when the number is supposed to be "zero". That's why, when we find a "zero" exponent, we ignore the mantissa altogether, knowing that the value of the whole thing is zero (to encode a zero simply make all bytes zero).

This encoding system has two constraints: the range of the values depends on the size of the exponent, and the accuracy of encoding depends on the size of the mantissa. In our computers the single-byte exponent allows up to 127 shifts either way. Thus the largest value that can be encoded is approximately 2 to the power of 127 and the smallest is 2 to the power of -127, or about 10 to the power of plus/minus 38 (1 with 38 zeros), the range of both our "single" and "double precision" numbers. The "single precision" numbers are encoded in 4 bytes. One byte is used by the exponent, leaving 3 bytes or 24 bits for the mantissa, enough to encode up to 7 decimal digits (the computer displays only 6, but we can always enter 7-digit values to increase the accuracy of the internal encoding). The "double precision" numbers use 8 bytes. Their 7-byte or 56-bit mantissa can encode up to 16 decimal digits, but only 15 are displayed. Thus, even though we can handle astronomic and subatomic quantities, only 6 or 15 digits are meaningful. The rest gets rounded off. Maybe it's just as well. We would go nuts if we had to deal with 38 digits.

Regardless of the "precision", most decimal fractions are encoded as approximations. Obviously only those fractions that are exact sums of negative powers of 2 can be encoded exactly, such as 0.75=1/2+1/4 (of course only if the number of digits does not exceed the "precision"). Other decimal fractions, when represented in terms of powers of 2, produce a binary mantissa which strictly speaking is infinite and at some point converges into a repetitive series. For example the innocent-looking decimal fraction 0.3 produces a binary mantissa in which a four-bit "nibble" 1001 repeats forever.....

.0001 1001 1001 1001 1001 .....

But in the computer the mantissa gets chopped down to fit in the available quantity of bits. When the first bit of the chopped-off "tail" happens to be "1" then the remaining mantissa is rounded up. Thus the 24-bit "single-precision" mantissa for decimal 0.3 looks like this, with the last "nibble" rounded up from 1001 to 1010.....

.0001 1001 1001 1001 1001 1010

These approximations and roundings are usually not a problem, as long as all computations are done consistently in the same precision. Using higher precision numbers in computations which are performed in lower precision (such as using double-precision variables in single-precision trig functions in BASIC) also works fine, because the computer already knows how to do the "chopping". The difficulty arises when lower precision fractions are forced into higher precision. The computer has no way of guessing what the continuation of the bit pattern should look like, so it blindly fills the additional mantissa bytes with zeros (another proof positive that humans are smarter than computers, in case you ever doubted it; we can often make such guesses, given the initial pattern of a series; it is a typical question in all IQ-tests). For example when the single precision representation of decimal 0.3 is converted to double precision, the mantissa becomes.....

.0001 1001 1001 1001 1001 1010 0000 0000 ...

Instead of continuing the "1001" pattern to the end, the conversion leaves the rounded "1010" in the middle, followed by meaningless 0's. This causes the well-known problem in BASIC when such conversions are done with the CDBL function or by direct assignment.....

```
A! = 0.3
A# = CDBL(A!)    .....or.....    A# = A!
```

The result is a strange-looking mess.....

PRINT A# displays .300000011920929

The first seven decimal digits are correct,

because that was the limit of encoding accuracy in the original single precision number. The rest is garbage, and if the variable A# is now used in double precision calculations, the error will distort all subsequent results. Fortunately in our "Microsoft" BASIC we have a way to get around it.....

```
A! = 0.3
A$ = STR$ (A!): A# = VAL (A$)
```

Now A# will be .300000000000000 (normally displayed as .3 unless you format it in PRINT USING with 15 decimal places), because the VAL function forces the computer to re-encode the complete binary series "from scratch". The rounding "nibble" 1010 is now properly placed at the end of the mantissa, instead of remaining in the middle as it did in the number-to-number conversion. You might think that this trick could have been easily built into the CDBL function, and you're probably right, but, as Hillary Rodham Clinton put it (in a different context), "coulda, shoulda, woulda, didn't".

If you'd like to take a look at the actual bit patterns of the "floating point" numbers in our BASIC's then try this simple program.....

```
11 clear 1000 """this line for Mod-I or III only
12 print "Single or double precision (S/D)? ";
13 line input k$: if k$="" then END
14 z%=instr("SsDd",k$): if z%=0 then 12
15 print "Enter number: ";
16 line input k$: if k$="" then 12
17 z!=val(k$): z#=val(k$): gosub 700

20 for x=1 to len(z$)
21 print mid$(z$,x,1);
22 if (x/8)=fix(x/8) then print " ";
23 next: print
24 goto 15

700 'FLOATSTR subroutine
710 z$="": b=3: v=varptr(z!)
720 if z%>2 then b=7: v=varptr(z#)

800 for y=0 to b: p = peek(v+y)
820 for x=1 to 8
830 k = p/2: p = fix(k)
840 if k>p then z$="1"+z$ else z$="0"+z$
850 next x, y
860 RETURN
```

The FLOATSTR routine evaluates either a single precision (z!) or double precision (z#) number, depending on the precision code (z%), and returns a string (z$) of 0's and 1's. The 800-series of lines is a loop that peeks at the bytes of the number, and extracts the bits from each byte by a repetitive division by 2 (discussed last time in TRSTimes 7.6). On return the string of bits is displayed in the 20-series of lines. To improve its legibility a space is inserted every 8 bits in line 22 (but if you use Model I or III for looking at double precision numbers then delete line 22, because the 64 bits with the extra spaces would exceed the screen width). And remember that the first 8 bits are the "exponent" byte.

REFILL YOUR DESKJET INK CARTRIDGES

by Doug Hyman

I have settled on the following procedure for successfully refilling my cartridges (after reading the compiled message threads in the CIS Deskjet Library and doing some additional experimentation.)

Materials: Syringe, 23 gauge 1" needle
Parker Quink Permanent Black Ink ($2.50/bottle at the Office Club). Quink has an additive that reduces clogging.

An empty ink cartridge weighs about 0.9 oz, and a filled one about 1.6 oz.

DIRECTIONS:

1. Fill the syringe with ink. As with any sharp object, handle it with great care.

2. Insert needle into the top hole in the cartridge. If you angle the needle towards the back of the cartridge you can insert it almost to the bottom.

3. _Slowly_ (I repeat...slowly) inject the ink in to the cartridge. If ink comes out of the hole you are injecting it too fast.

4. Weigh the cartridge at intervals (I use a small hand-held postal scale) so its re-filled weight with ink is about 1.6 oz.

Note: It appears from the message thread, and my own experience, that it is best to refill the cartridge to a weight of at least 1.5 oz.

Remember when refilling that the inside of the cartridge has delicate electrical connections that can be broken with careless use of the syringe. In addition, injecting the ink too quickly can increase internal pressure to levels that will damage the cartridge.

When refilling it pays to have a new unused cartridge available in case the refill attempt fails.

When finished, wipe ink from the cartridge, the jets, and electrical contact surfaces. Use a soft tissue or paper towel wadded up and moistened with water. Brush the surface lightly with the tissue, and pat the surface dry. Do not try to remove all the ink from the surface of the jets. The idea is to remove excess ink and stop clogs.

Install the cartridge and run the printer self-test. The line at the top is a test printing of each jet. If the line is uneven in intensity, or there are gaps, then the jets need cleaning.

When finished reinking rinse the syringe and needle immediately with water, put the protective cap back on the needle, and store them. The cartridges I have refilled with this method have worked immediately. While one or two jets show signs of clogging when first tested, removing the cartridge and cleaning the jets gets them going again. Print quality is comparable to a new HP cartridge, though the ink takes slightly longer to dry.

Clogging problems usually appear when [1] the jets clog on a new or refilled cartridge when it is removed from the printer, [2] the ink formulation dries in the cartridge (here Parker Quink has an additive to prevent this), and [3] not enough ink is added to the cartridge and the ink does not wick through to the jets, and [4] a filled cartridge is left out without being sealed and protected, permitting air to dry out the jets, and the internal ink.

STORAGE:

When storing cartridges to be refilled, or after refilling, you must seal them to prevent air from entering.

On opening a new HP ink cartridge I save the empty cartridge box and the tape that protects the jets. When I refill a cartridge, I clean it off, test it out, reclean the jets, cover them with the protective tape, and wrap it well in a plastic wrap. It is then put in a plastic sandwich bag, sealed tightly, and stored in the original box.

If you want to save an empty cartridge until you reink it, store it the same way to prevent it from drying out.

Good luck on your attempts to reink.

HINTS & TIPS

TRANSFERRING TRSDOS 1.3 FILES

by Kelly Bates

Have noticed discussion on transferring files from TRSDOS 1.3 to other places. I use NEWDOS/80 to do this - my method is as follows:

Boot NEWDOS/80, type PDRIVE 0 and select the following pdrive spec as drive 1

TI=AM, TD=E, TC=40, SPT=18, TSR=3, GPL=6, DDSL=17, DDGA=2

This format is usually number 4 on the pdrive listing. So the command then would be PDRIVE, 0,1=4,A. Pressing <ENTER> will make the change in pdrive occur.

Now issue this command:

COPY,1,0,09/26/93,NMFT,USR,CBF,CFWO,NDMW <ENTER>

You will then be selective in which files you copy to drive 0. The command parameters indicate 'date, no format, USeR files only, Copy By File, Check File With Operator, No Disk Mount Waits'. You can copy with greater or fewer parameters, but this one works fine.

After the files are copied, remove the TRSDOS disk from drive 1.

The next step is usually to copy the files to a single-density disk, so set the pdrive as follows to format a single density disk in drive 1:

PDRIVE,0,1=5,A <ENTER>

It should read:

TI=A, TD=A, TC=40, SPT=10, TSR=3, GPL=2, DDSL=17, DDGA=2.

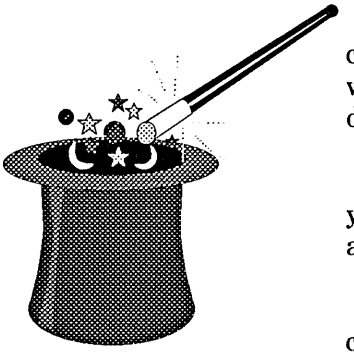
Now format a disk in drive 1 with the command

FORMAT :1 <<ENTER>>

After the format is complete, copy your files to the disk in drive 1. You can now move the files anywhere you want to. I do this routinely as I move fonts around.

MODEL 4 CARD TRICK

By James Sowards



Hi, I'm your Model 4 computer! People are always asking me the dumbest questions....

Can you do this? Can you do that? Can you do another thing?

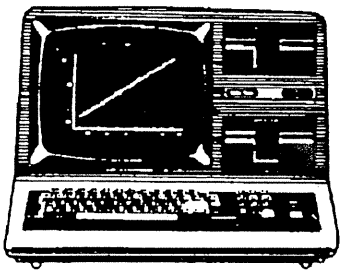
The answer is YES! I can do all of those things. I can do Wordprocessing... I can do Databases... I can do Spreadsheets... I can play games... I can even do card tricks.

Wanna see a card trick? Good...

You will be given three rows of cards... choose a card and tell me which row it is in 3 times, and I will tell you which card it is!!!

```
10 'cardtrk/bas
20 'by James Sowards
30 '
40 '
50 DATA ACE,TWO,THREE,FOUR,FIVE,SIX,
SEVEN,EIGHT,NINE,TEN
60 DATA JACK,QUEEN,KING
70 '
80 '
90  REM ROUTINE TO MAKE THE DECK -
DECK$(I,J)
100 '
110 DIM DECK$(4,13),CARD$(21)
120 FOR I=1 TO 4
130 FOR J=1 TO 13
140 IF I=1 THEN SUIT$=" OF HEARTS"
150 IF I=2 THEN SUIT$=" OF DIAMONDS"
160 IF I=3 THEN SUIT$=" OF CLUBS"
170 IF I=4 THEN SUIT$=" OF SPADES"
180 READ A$
```

```
190 DECK$(I,J)=A$+SUIT$
200 NEXT J
210 RESTORE
220 NEXT I
230 '
240 FR$=STRING$(64,140)
250 '
260 CLS
270 PRINT CHR$(15)
280 '
290 REM SELECT CARDS FROM DECK AT
RANDOM AND REPLACE WITH `-1'
300 REM TO REMOVE THEM FROM THE DECK
SO THEY CAN'T BE SELECTED AGAIN
310 '
320 RANDOM
330 FOR C=1 TO 21
340 A=RND(4)
350 B=RND(13)
360 IF DECK$(A,B)="-1" THEN 340
370 CARD$(C)=DECK$(A,B):DECK$(A,B)="-1"
380 NEXT C
390 '
400 REM TO FORMAT THE CARDS IN THREE
ROWS AND PRINT
410 '
420 K=0
430 FOR I=1 TO 19 STEP 3
440 K=K+1
450 ROW1$(K)=CARD$(I):
ROW2$(K)=CARD$(I+1):ROW3$(K)=CARD$(I+2)
460 NEXT I
470 K=0
480 CLS
490 '
500  REM TO PRINTOUT THE RESULTS
510 '
520 PRINT@(3,7),FR$:PRINT
530 PRINT TAB(8)"ROW ONE";
TAB(32);"ROW TWO";TAB(55);"ROW THREE"
540 PRINT TAB(8) FR$
550 PRINT
560 FOR I=1 TO 7
570 PRINT TAB(8) ROW1$(I);TAB(32);ROW2$(I);
TAB(55);ROW3$(I)
580 NEXT I
590 PRINT TAB(8)FR$
600 PRINT
610 PRINT TAB(8)" --> Which row is the card in (1-
2-3)....";CHR$(14);
620 INPUT RW
630 PRINT CHR$(15)
640 COUNT=COUNT+1
650 ON RW GOTO 680,780,880
660 GOTO 400
670 '
680  REM FOR ROW ONE
690 '
700 FOR I=1 TO 7
710 CARD$(I)=ROW2$(I)
720 CARD$(I+7)=ROW1$(I)
730 CARD$(I+14)=ROW3$(I)
740 NEXT I
750 IF COUNT=3 THEN 980
760 GOTO 400
770 '
780  REM FOR ROW TWO
790 '
800 FOR I=1 TO 7
810 CARD$(I)=ROW3$(I)
820 CARD$(I+7)=ROW2$(I)
830 CARD$(I+14)=ROW1$(I)
840 NEXT I
850 IF COUNT=3 THEN 980
860 GOTO 400
870 '
880  REM FOR ROW THREE
890 '
900 FOR I=1 TO 7
910 CARD$(I)=ROW2$(I)
920 CARD$(I+7)=ROW3$(I)
930 CARD$(I+14)=ROW1$(I)
940 NEXT I
950 IF COUNT=3 THEN 980
960 GOTO 400
970 '
980  REM TO DISPLAY THE CORRECT CARD
990 '
1000 COUNT=0
1010 PRINT:PRINT TAB(8)"... AND YOUR CARD
IS THE ";
1020 '
1030 FOR I=1 TO LEN(CARD$(11))
1040 PRINT MID$(CARD$(11),I,1);
1050 FOR J=1 TO 200:NEXT J
1060 NEXT I
1070 PRINT:PRINT
1080 PRINT@(22,7),"Want to try again...
(Y/N)";CHR$(14);:INPUT Z$
1090 PRINT CHR$(15);
1100 IF Z$="N" OR Z$="n" THEN CLS:END
1110 IF Z$="Y" OR Z$="y" THEN 320
1120 GOTO 1080
```



SOME HISTORY ON THE 8080 AND Z-80 CPUS

by Roy T. Beck

I was thinking recently about the Z-80, the heart (brain, really) of most of our TRS machines, and thought I might write a little about its history, capabilities, and what ever else came to mind. This is the result.

The Pedigree of the Z-80

Like much of industry and technology, the Z-80 did not just suddenly appear, without parents, history, etc. Instead, it is the result of a series of technological and commercial developments over a period of time, and, I believe, it is still going on.

Its oldest ancestor, that I am aware of, was a 4 bit chip known as the 4004 which was originally dreamed up as a calculator chip. That was in the days when a 4 function desk calculator went for several hundred dollars, and was a true technological marvel of its day, which was around the 1960's if I remember correctly. The 4004 was superseded by an 8 bit chip identified as the 8008. I don't know what applications there were for that chip. Finally, Intel (of Pentium fame) created the 8080 8 bit chip, which was the chip chosen by the early kit builders and Digital Research who created CP/M. The 8080 chip was logically sound, but had some peculiar mechanical and electrical quirks.

For reasons I don't know, a principal (maybe chief) designer of the 8080 chip left Intel and opened up a competing operation known as Zilog. At Zilog, he developed our dear Z-80, since he had in his head a complete understanding of the 8080, along with knowledge of its rough edges. It's mechanical and electrical interface was simplified and improved over that of the 8080. The Z-80 had many new features, but was intended to be completely upwardly logically compatible from the 8080. There is a small tickle in my memory which reminds me there is a wrinkle in the Z-80 where a flag was handled slightly differently, which caused some trouble in CP/M when that was THE operating system. A one byte correction was sufficient to correct it, and the whole problem faded into history.

The last widely known successor to the 8080 was an Intel chip known as 8085. This chip had the same instruction set as the 8080, but with two additional instructions. These two were RIM and SIM. Since they were bit mapped, they were effectively many instructions, and added a lot of I/O capability to the chip. RS used the CMOS version of this chip in the Model 100/102 family, with good results. The CMOS construction greatly reduced power consumption, making battery operation more feasible than if TTL design was used.

There is also supposed to be a Z-800 and Z-8000 version of the Z-80. I am sure the Z-800 exists and is available, but I know nothing of it, and I am not sure the Z-8000 ever got past the public rumor stage.

The Instruction Set

The 8080 chip was equipped with a set of one byte instructions (Opcodes), some of which expected to be followed by one or more operands. The fact that all the opcodes were one byte long meant that there could only be 256 opcodes in the instruction set. This is all the possible values in an 8 bit byte. Actually, the 8080 only had 244 opcodes. The following values were not valid opcodes:

08, 10, 18, 20, 28, 30, 38, CB, DD, D9, ED, FD.

While I am not sure of this, I am reasonably confident these were handled internally as No Ops, (NOP), so you could really say the 8080 had 13 NOP codes instead of the one official value of 00.

Anyway, The Z-80 came along, complete with 696 opcodes. With only 12 unused opcodes, how did they add 452 more? The answer was two fold. First, 8 new one byte opcodes were added, leaving 4 unused values, CB, DD, ED, and FD. These 4 values were internally defined to be access ports into a family of 2 and 3 byte opcodes. Of course adding opcodes in this fashion increases the execution time of some of the instructions, but it was a workable way of increasing the size of the instruction set.

What kinds of instructions were added? The 8080 completely lacked relative jumps, so 5 opcodes were added.

JR	Jump Relative, Unconditional
JRZ	Jump Relative, Zero
JRNC	Jump Relative, Non-Zero
JRC	Jump Relative, Carry
JRNC	Jump Relative, No Carry

Unfortunately, the range of the Relative Jump is limited to +127 and -128. This is certainly limiting, but is far better than not having JR at all.

Another powerful instruction, DJNZ was added, to facilitate looping. Again, the range is limited to 256 loops, but this works well for many applications. With some trickery, the programmer can easily extend this range.

Another major feature of the Z-80 is its Alternate set of registers, about which I will say more later. Two toggle-type instructions were added to facilitate swapping the registers, and these are EX AF,AF', and EXX.

One of the features of the Z-80 is its ability to manipulate bits in registers and memory locations. The generic instructions are SET, RESET and BIT. SET will force a 1 in a given bit position, RESET will force a 0, and BIT will report the present status of a bit without altering it. For some purposes, such as graphics, this opened up a whole new playing field! Yes, I know you can set bits by means of bit masks with ANDs, ORs, XORs and the like, but this allows much more flexible operation.

The Z-80 is organized around the concept that the world consists of seven 8 bit registers A, B, C, D, E, H, and L, and one memory location, pointed to by the HL register. On this basis, 8 register or memory cells times the 8 bits in a byte times 3 instructions equals 192 opcodes. Since each of the 4 "holes" in the 8080 instruction set can accommodate 256 two byte instructions, all 192 bit manipulation instructions can easily fit in one hole, in this case the CB opcode. In fact, 256 less 192 leaves 64 more.

Another useful type of instruction is the bit shift and rotate type. I am no expert on these, so I will just remark that a batch of these were also tucked into the CB hole.

Two other new registers, IX and IY were added in the Z-80. These are useful in indexed operations where the IX or IY instruction identifies the beginning of a table, for example, and a following one byte operand can signify offsets from the location where the IX or IY is pointing. More than 40 instructions were added for each of IX and IY. These instructions are available in matching pairs, one for IX and one for IY in each case. To accommodate these, the holes at DD and FD were assigned. All the IX instructions are in the DD hole, and all the IY instructions are in FD.

Finally, hole ED was still available. This appears to have been used as a "catchall" for an assortment of other features. Some of the items included here are some additional IN and OUT instructions, and most importantly, some block move and search instructions, notably including LDIR.

Also included were some additional 16 bit load instructions, and finally some additional Interrupt handling capabilities.

Even some three byte instructions were added, and these had to be accessed through two layers of "holes". These additional instructions pertain to the IX and IY opcodes, which means accessing them through either DD or FD, and then through a hole CB in those plains to the third byte of the opcodes.

Undocumented Opcodes

Officially, only 696 instructions exist in the Z-80. However, clever programmers discovered some other 2 and 3 byte instructions appeared to work reliably. Most of these weren't particularly useful, but a couple were. I won't detail these, as I don't know them, and of course they don't exist in the official documentation. It is interesting to note that these same "undocumented" opcodes appear in Mostek's version of the Z-80, which was a second source to Zilog. My guess is that Mostek simply used copies of Zilog's photo mask drawings in order to prevent any inconsistencies and to make them a reliable second source. All of this is supposition on my part, so accept it on that basis. Anyone seriously interested in the undocumented opcodes can find them written up, I believe, in 80 Micro magazine.

There is an interesting story to be told about one of the undocumented opcodes. A company named Hitachi much later brought out a "super" version of the Z-80 with some additional instructions and a greater clock speed. It was named HD64180. They did not copy the Zilog microcode; instead, they appear to have started with a clean sheet of paper, and designed from the ground up, following the "official" Z-80 instructions. This had two effects. Those few adventuresome individuals who were making use of the undocumented Z-80 opcodes suddenly found their code would no longer execute when run on a machine equipped with the HD64180 chip. Of course Hitachi and RS simply ignored this whole episode,

on the basis that no one had any business attempting to use undocumented instructions. Now that was probably an adequate answer for the peasants, and could well have been the end of that story.

However, a few adventuresome souls installed HD64180's in their Models 3 and 4. This indeed gave a nice speed increase to the Model 4's running LS-DOS, but lo and behold, the Model III and the Model 4 in the III mode simply refused to boot TRSDOS V 1.3. WHA HOPPEN?? The answer is that the anonymous programmer who wrote TRSDOS V 1.3 had elected to use an undocumented opcode in his boot loader, and the HD64180 choked on the "illegal" opcode and hung. Why did RS do that? I never heard an answer to that question. What was to be done about it? Turned out that a one-byte patch (the best kind) could fix the problem, and life went on, with HD64180's clocking merrily (and rapidly) along in some owner's machines. But I think it is hilarious that good old RS should get caught doing something like using illegal (undocumented) opcodes! The problem came about, of course, because Hitachi followed literally and strictly the published Z-80 instruction set, no ifs, ands, or buts.

Probably we are all very lucky that Mostek built in the same, identical real opcodes that Zilog had designed in. Imagine the confusion if we had to buy programs, or write them, on the basis that some users had slightly different CPU's than others.

That's not the only example of such deviations. Radio Shack has used two different hard drive controllers in the TRS series over the years, and the two boards are NOT logically identical. Almost, mind you but not quite. I got burned on the difference once in dealing with NEWDOS/80, but that's another story.

Disassemblers

As you all know, we have a number of good disassemblers available for Z-80 code. An important point is that these disassemblers should automatically synchronize with the code. That is, the disassembler should be able to get into step with the code it is being fed, and should recognize an 04h, for instance, as the instruction INC B, and not mistake it for an operand of some other opcode, or as part or a multiple byte opcode. While not perfectly self synchronizing, the Z-80 disassembers are very good at self synchronizing, largely due to the predominance of one byte opcodes. In the simplest case, if there were no operands, no ASCII data and only one byte codes, the disassembler could never get out of synchronism. With mostly one byte opcodes, and the disassembler designed to attempt to force every byte as an opcode unless an operand or second byte is required by the previous byte, the problem of synchronization is seldom much trouble.

Some, but not all disassemblers print the ASCII value of every byte, if there is one, on the same line as the source code in the printout.

One feature that I greatly enjoy in the disassembler included in NEWDOS/80 is a reverse reference table. This feature will point out every CALL, JUMP, etc throughout your code which refers to any given address anywhere in your machine's memory map. It also identifies any instruction which loads, for example a one or two byte address. If there is an instruction which loads an "8" into a register or memory location, the reverse reference table will give you the address of every instruction which does this. Another useful feature is its ability to identify all instructions which reference a particular port. Need to find all references to your printer port? They are all shown.

Where code contains a block of ASCII code which was intended to be displayed as a message on a screen or printed out, the disassembler will attempt to disassemble all the ASCII values as one byte opcodes. The result, for a Z-80, is a succession of Loads, most of which make no logical sense when analyzed. Of course, if the disassembler simultaneously prints out the ASCII values, then the ASCII text can be read vertically down the page, and in such cases the opcodes can be ignored as meaningless.

Silly Instructions

For simplicity, the 8080, and thus the Z-80 did not suppress some of the instructions which showed up in the pattern of opcodes. An instruction such as LD B,B is just such a silly instruction. It means move the contents of the B register into the B register, which is a useless move. In the design of the 8080, there are a number of such moves. They could have been suppressed at design time, but this would have increased the design effort, and raised the design cost. The result is that such silly opcodes exist, and are simply to be tolerated. A knowledgeable programmer knows to expect them in a disassembly, and ignores them in most cases. I say most, because there is one possible legitimate use for such codes; they may be used as deliberate time wasters, where a small time delay is needed for synchronization purposes between chips.

Missing Instructions

From a programmers point of view, there are some instructions missing from the Z-80 which would have been nice to have, but sorry, they are not there! An example is the shortness of range of the relative jumps. The limit of +/- 128 bytes is sometimes a pain, especially if you want to create portable code which can be loaded and made to run anywhere in a Z-80 machine. Since there is no way to relative jump more than the +/- 128 rule allows, any greater jump must be hard coded with absolute addresses in order to function. If you the programmer don't know where the code is to be loaded, then you must do some clever tricks to find out where your routine got loaded, adjust your hard coded jumps, calls, etc to suit, and then go about your business. This can be and is done, but it is a nuisance which could have been avoided if the chip had a longer relative jump range.

A funny story about RS again on just this point. RS created an assembler which was first sold as a tape version for the Model I, and later was offered as a disk version for disk machines. When they wrote the assembler, they forgot to check the code to be assembled for the length of the relative jumps. If the jump was short, no problem; if the jump was long, the assembler would truncate a high bit or two, and produce inoperable relative jump codes, and not raise a flag! There were at least two versions of the original assembler released with this error. I assume RS eventually fixed it, but they turned out mucho defective code at various times, usually saying nothing until challenged by a knowledgeable user, and sometimes not then. These were sometimes facetiously known as "undocumented features" in bull sessions at clubs and restaurants.

Interrupt Handling

Another great feature of the Z-80 is its greatly increased interrupt handling capabilities, relative to the 8080. As you know, an interrupt can be used to cause the CPU to drop whatever it is currently doing and undertake a high priority task, which might be a real-time task, such as controlling the temperature of a steam boiler, or the speed of a turbine which must not be allowed to overspeed. In fact, the Z-80 is equipped to handle a whole series (up to 8, I believe) of high priority tasks, handling them in sequence of their priority, which must be assigned by the programmer. A further capability, in interrupt mode 2 is for the Z-80 to fetch a portion (one byte) of the interrupt handling code address from the computer's bus, supplied by some other hardware of the computer.

The Z-80 was never designed just for our personal computing amusement; it was designed as a workhorse industrial chip, and it is widely used in the industrial world. It is for such reasons of overkill, as far as we are concerned, that some of its features are not used in our machines. In fact interrupt mode 2 is not available to us, as some of the required hardware does not exist in our machines.

The Alternate Register Set

Getting back to high speed, high priority interrupt processing, it is always possible to "save the environment" when interrupt duty calls by poking all the register contents to memory somewhere, shift over to the interrupt handler code, do whatever is necessary, and then come back to the interrupted foreground task. But all the registers would have to be reloaded before the task could proceed. But the Z-80 to the rescue! The Alternate Register set I mentioned much earlier can handle this type of situation. The Z-80 contains a duplicate set of A, F, B, C, D, E, H, and L registers. These are identified in the manuals as A', F', etc, all with superposed apostrophes. How are they used? The two extra instructions EX AF,A'F' and EXX are simple toggles which swap access from AF to A'F', for example, and the other instruction swaps the BC, DE, and HL pairs en masse in one transaction. The reason for this is that the interrupt handler can count on the alternate register set being preloaded with necessary values required to service the high speed interrupt task without having to save the environment before proceeding.

I have never written code using the alternate register pairs, at least not on the Z-80. However, my programmable HP-97 desk top calculator has a similar setup, and I did make use of it in one program where I needed more registers than were available. The problem I ran into while debugging the program was to keep track of which set of registers were "up" at any given moment. I made it work, but I had the same problem there as I would have in the Z-80; no flag to show whether primary or alternate registers are presently in use.

Probably for just this reason, all Radio Shack DOS and application programmers have made it an inviolate rule; "Don't use the Alternate Registers"!

Restarts

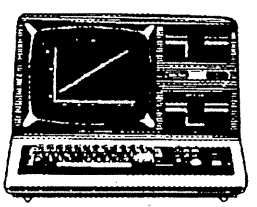
A restart is a funny kind of Call instruction, although I have never seen it described that way. It is a one byte instruction which, like a call, saves the current value of the Program Counter (PC) on the

stack, and like a call jumps to a specific location in memory. The difference is that you don't tell it where in memory to jump to; it can only go to one place, by definition of the microcode in the CPU, and therefore is a one byte Call, instead of using three bytes as normal Calls do. It is terminated by a Return, just as with any other Call. TRSDOS 6.X uses Restarts extensively to conserve code. Go disassemble and read some of Roy Soltoff's code if you want to improve your own.

Another feature added to the Z-80 was Restarts 38h and 66h. Both of these are peculiar for the reason that they are not opcodes, but are the result of momentarily grounding their respective terminals on the Z-80 chip itself. One terminal is identified INT*, meaning "Interrupt". This one is maskable, meaning it can be turned off in software. When it is hot, it produces Restart 38. This is one of the primary means of introducing high priority interrupts into the machine from external circuitry, etc. The other one is identified as NMI*, and activates Restart 66h. The NMI means "non-maskable interrupt", and cannot be turned off in software. This one obviously has greater priority than INT*.

As you may know, the "Reset" button on our machines does not connect directly to the RESET pin on the CPU. On the Model I and III it went to the NMI* pin, which caused it to go through some software linkages before it causes the Z-80 to jump to 0000h, mainly to avoid resetting pointers which will cause everything to cold start. For most purposes, this is OK, as it allows the firmware designer of the machine to design explicitly the response of the machine to the Reset button. I believe the Model 4 functions the same way, but I haven't investigated it. This type of "Reset" offers surgical precision, as opposed to a hammer blow, you might say. And that's all well and good until a runaway program destroys part of the Reset software chain. Then you can push the Reset button futilely until your digit falls off, and no useful result obtains. Turn the power switch off, and start over. Sigh.....

I am sure there are many more interesting and/or humorous stories about the Z-80 and its kin in existence, but I have just about used up my quota. Accordingly, I will sign off here and say I have enjoyed the telling of what I know, and hope you have found a bit of pleasure in the reading of it.



C Programming Tutorial

Part 5

by J.F.R. "Frank" Slinkman

In the last chapter, I asked you to think about how recursion could be used to program a computer to play a perfect game of checkers.

To illustrate the concept, let's tackle a much simpler game -- one of those annoying "keep you occupied so you won't notice how slow the service is" games you find in many restaurants and truck stops.

Specifically, the one with 15 holes arranged symmetrically within a triangle and 14 pegs or golf tees, with the object being to jump and remove pegs checkers-style until only one peg remains, preferably in the hole which was initially empty.

The program pegwins.c is both overkill and underkill at the same time. It's underkill because it does not attempt to find ALL the possible solutions -- just one solution for each starting point.

Finding all of them would take a lot longer, and most of the extra solutions would merely be reflections or rotations of previously found solutions.

If you'll look at the map of the holes in the listing below, you'll see there are only four unique starts, namely with one of holes 0, 1, 3 or 4 empty. All other starts are rotations or reflections of one of those four situations.

Thus pegwins.c is overkill because finds a solution for each of the possible 15 starts, meaning many of the solutions it finds are just reflections or rotations of previously found solutions.

You'll notice that in this program some of the variable names are in upper case letters. If you'll think back to the very first article in this series, you'll remember I mentioned there were some commonly used conventions regarding capital letters.

These are:

Defined variables such as these (MAP, SORC, JUMP, DEST, DO and UNDO) are often in caps; and Constant expressions, whose values are not to be changed by the program, often have their first character in caps. For example:

```
long    Ten_thou = 10000L;
double  One_third = 1.0 / 3.0;
```

Now let's let the computer solve that annoying peg game for us once and for all:

```
/* pegwins.c */

#include  <stdio.h>
#option INLIB
#option ARGS      OFF
#option FIXBUFS   ON
#option MAXFILES  0

#define MAP    struct map
#define SORC   0
#define JUMP   1
#define DEST   2
#define DO     1
#define UNDO   0

void   get_best(), report(), find_moves();
void   do_move();

char   jumps[][3] =
/*
 *            0
 *          1  2
 *        3  4  5
 *      6  7  8  9
 *    10 11 12 13 14
 *
 *     SORC over JUMP to DEST
 */
{  { 0,  1,  3 }, { 0,  2,  5 },
   { 1,  3,  6 }, { 1,  4,  8 },
   { 2,  4,  7 }, { 2,  5,  9 },
   { 3,  1,  0 }, { 3,  4,  5 },
   { 3,  7, 12 }, { 3,  6, 10 },
   { 4,  7, 11 }, { 4,  8, 13 },
   { 5,  8, 12 }, { 5,  4,  3 },
   { 5,  2,  0 }, { 5,  9, 14 },
   { 6,  3,  1 }, { 6,  7,  8 },
   { 7,  4,  2 }, { 7,  8,  9 },
   { 8,  7,  6 }, { 8,  4,  1 },
   { 9,  8,  7 }, { 9,  5,  2 },
   { 10,  6,  3 }, { 10, 11, 12 },
   { 11,  7,  4 }, { 11, 12, 13 },
   { 12, 11, 10 }, { 12,  7,  3 },
   { 12,  8,  5 }, { 12, 13, 14 },
```

```
  { 13, 12, 11 }, { 13,  8,  4 },
  { 14, 13, 12 }, { 14,  9,  5 } };

char   Header[] = { "\
       A\n\
     B  C\n\
    D  E  F\n\
   G  H  I  J\n\
 K  L  M  N  O" };
char   Border[] =
     { "=========================\n" };

int    pegs;
char   best[13];

MAP { char   map[15];
      char   move[36];
  } master;

/*=*=*=*=*=*=*=*/

main()
{
   static int  i;

   puts( Header );
   puts( Border );

  for ( i = 0; i < sizeof master.map; )
  {  memset( master.map, 1, sizeof master.map );
      master.map[i] = 0;
      printf("Start with hole %c empty\n\n",
              'A' + i++ );
      get_best( &master, best );
      report();
   }
}
/*=*=*=*=*=*=*=*/

void get_best( old_map, bst_ptr )
MAP *old_map; char *bst_ptr;
{
   MAP    new_map;
   int    i = 0;

   memcpy(&new_map, old_map, sizeof(MAP));
   find_moves( &new_map );

   while ( i < sizeof new_map.move )
   {  if ( new_map.move[i] )
      {  *bst_ptr = i;
         do_move( &new_map, i, DO );
         if ( !( pegs = count_pegs(&new_map) ) )
            break;
         else
         {  get_best( &new_map, bst_ptr + 1 );
            if ( !pegs )
               break;
            else
               do_move( &new_map, i, UNDO );
         }
      }
      ++i;
   }
}
/*=*=*=*=*=*=*=*/

void report()
{
   static int  i;
   static char *ptr;

   for ( i = 0; i < sizeof best; )
   {  ptr = jumps + best[i++];
      printf( " Jump %c over %c to %c\n",
              'A' + *(ptr + SORC),
              'A' + *(ptr + JUMP),
              'A' + *(ptr + DEST) );
   }
   puts( Border );
}
/*=*=*=*=*=*=*=*/

void find_moves( map )
MAP *map;
{
   static int  i;
   static char *ptr;

   memset( map->move, 0, sizeof map->move );

   for ( i = 0; i < sizeof map->move; )
   {  ptr = jumps + i;
      if (  map->map[ *(ptr + SORC) ]
        && map->map[ *(ptr + JUMP) ]
        && !map->map[ *(ptr + DEST) ] )
          map->move[i] = 1;
      ++i;
   }
}
/*=*=*=*=*=*=*=*/

int count_pegs( map )
MAP *map;
{
   static int  i, count;

   for (count = -1, i = 0; i < sizeof map->map;)
      if ( map->map[i++] )
         count++;

   return
}
/*=*=*=*=*=*=*=*/
```

```
void do_move( map, move, code )
MAP *map; int move, code;
{
   static char *ptr;

   ptr = jumps + move;

   map->map[ *(ptr + SORC) ] =
   map->map[ *(ptr + JUMP) ] =
   !( map->map[ *(ptr + DEST) ] = code );
}
```

Note we haven't "optioned" out standard I/O redirection, since you probably will want to send the output to your printer.

The first "define" is just a way to save some typing and keep the listing clearer and simpler. Defines are straight text substitution macros; so all we've done here is tell the compiler to replace every occurrence of "MAP" with "struct map."

The next three defines are to help keep the listing easier to understand. "SORC," "JUMP," and "DEST" will be used to reference the array elements which describe the nature of each possible move.

DO and UNDO are switches for the do_move() function -- again, because the words make the logic of the program easier to follow for someone reading the listing than would the values 1 and 0.

After the prototypes, we declare and initialize a multi-dimensional array. Each of the elements in this array describes one of the 36 possible moves on the board.

But note its declared "jumps[][3]" instead of "jumps[36][3]." How can we get away with that?

Remember, "jumps" is just a pointer to the first byte of the first element. The fact there are two sets of brackets tells the compiler the array has two dimensions, and the definition "[3] tells it each element has three bytes.

The fact is that the compiler doesn't CARE how many 3-byte elements "jumps" has. It's up to you, the programmer, to make sure no access to non-existent elements is attempted.

This is both an advantage and disadvantage for the programmer. It's an advantage because it lets us play with the indexing of arrays, such as we did with the "place" variable in prog06.c in Part 3.

It's a disadvantage because it makes it harder to track down errors. BASIC, for example, throws up a "subscript out of range" error if your program tries to go outside the defined range. C doesn't.

As with everything else in life, greater freedom means greater responsibility.

Also note how the array is initialized. The outside set of brackets contain the entire universe of values, while each inside set of brackets contains three values, one for each of the 3-char elements.

Next we declare two literal constants, "Header" and "Border." All literals are stored in permanently assigned RAM, regardless of their class; so there's no additional RAM cost to declare them as globals.

I just felt it made the listing clearer to declare them this way.

But note their lengths aren't contained within the brackets. Again, the variables are pointers; so the system doesn't have to know the length of what is being pointed to. And the bytes to store in RAM have been made obvious to the compiler through the use of braces and quotation marks.

The "pegs" variable is used to hold the count of the number of pegs on the board. The "best[]" array is used to record a sequence of the moves described by elements of "jumps[]."

Remember, the game begins with 14 pegs; each move requires the removal of one peg; and, at minimum, one peg remains at the end of the game. Thus the maximum possible number of moves in a game is 13 -- hence "best[13]."

"Master" is a struct of type map. But we didn't have to type "struct map" because of the definition of MAP.

Structs of type map contain two char arrays:

"map[]" to hold the locations of the pegs remaining on the playing board; and

"move[]" to hold the available moves for that specific situation. Each of the 36 move[] elements relates, by position, to a 3-byte jump[] element.

Now, in the main() function, notice the declaration of "i" to be of type int and class static.

But why use a static when we don't need one, which is contrary to "good" C programming practice?

The answer is program efficiency. First, statics

and globals are accessed faster than either autos or register variables. Second, statics require less code for access. This is not unique to the TRS-80 or Pro-MC, but is pretty consistent for all processors.

On our systems, accessing a register int requires code like:

```
PUSH    IX      ;the register variable
POP     HL      ;get value into HL
```

Accessing an auto int requires code like:

```
LD      HL,4            ;offset from SP
ADD     HL,SP           ;HL points to value
CALL    @GINT##         ;get value into HL
```

And the @GINT routine will be something like:

```
@GINT:
 LD     A,(HL)          ;p/u LSB
 INC    HL
 LD     H,(HL)          ;p/u MSB
 LD     L,A             ;value now in HL
 RET
```

But accessing a static or global only needs:

```
LD    HL,(I$)           ;get value into HL
```

The amount of code (3 bytes) to access a static or global is the same as is needed to access a register variable, but it's 56% faster -- 16 T-states vs. 25 on a standard Z80 or Z80a.

But look at all the code which has to be executed to access an auto integer.

Seven bytes worth of instructions vs. three (the @GINT routine appears only once; so we won't count the 5 bytes in that routine), and 72 T-states vs. 16!

More than twice as many bytes of code, and 4-1/2 times as long to execute!

The effects are compounded when you actually DO something with the variable -- for example, just incrementing a counter:

Class register:

```
PUSH    IX      ; 2 bytes       15 T
POP     HL      ; 1 "           10
INC     HL      ; 1              6
PUSH    HL      ; 1             11
POP     IX      ; 2             14
                 ;==            ==
                 ; 7 bytes      56 T
```

Class auto:

```
LD      HL,4    ; 3             10
ADD     HL,SP   ; 1             11
PUSH    HL      ; 1             11
CALL    @GINT## ; 3             17
;@GINT overhead                 34
INC     HL      ; 1              6
POP     DE      ; 1             10
CALL    @PINT## ; 3             17
;@PINT overhead                 62
                 ;==            ==
                 ;13 bytes      178 T
```

Classes global and static:

```
LD      HL,(I$) ; 3             16
INC     HL      ; 1              6
LD      (I$),HL ; 3             16
                 ;==            ==
                 ; 7 bytes      38 T
```

For the reasons illustrated above -- compactness and speed -- I usually make virtually all variables in my programs either globals or statics.

Incrementing the register variable could be MANUALLY optimized to INC IX (2 bytes, 10 T) before final compilation. But manually searching the compiler-produced assembly language listing to make such changes is a royal pain, and adds so much programming time it's highly unlikely you'd ever realize a net time saving. I suggest you forget this kind of "efficiency overkill" unless you're producing a pro-gram you intend to market commercially.

Also, there's a portability drawback with register variables. The 80x86 CPUs used in IBM-compatibles are so register-poor that implementations of C for those machines have to convert variables of class register to class auto. Thus the speed advantage of register variables on good computers (mainframes, 680x0-based machines like the "Mac" and, of course, the TRS-80) is lost when the code is ported to the MeSs-DOS environment.

Again, this is not "good programming practice;" so you might not want to make such extensive use of statics if the "political correctness" of your code is a consideration. (The thought police are everywhere! Now we even have PC on our PCs!)

The rest of main() is pretty straightforward. After displaying explanatory information, it merely initiates 15 games, each with a different hole empty at the start, and causes the results to be output.

The sexy part of the program is the get_best() function. This is a "brute force" recursive function which goes through every possible combination of moves until it finds one which results in a win (i.e., only one peg remaining on the board).

Because get_best() is recursive, it's variables must be of class auto. This means at every level of recursion, they are created anew on the stack -- even the 51-byte struct "new_map."

Note that in pure K&R C, such as ours, UNIONS AND STRUCTS CANNOT BE PASSED AS ARGUMENTS TO FUNCTIONS, nor can they be returned. But POINTERS to unions and structs can be freely passed back and forth.

ANSI C allows structs and unions to be passed to and from functions on the stack. But creating copies of large data structures on the stack can be quite slow; so should be done only if the original has to be absolutely bulletproof (i.e., no chance can be taken that the original struct or union will be altered by the called function).

Look back at main() to see how get_best() was originally called. The arguments for "old_map" and "tmp_ptr" were the address of (i.e., a pointer to) the struct "master" and "temp," which is really a pointer to the first byte of temp[].

Because get_best() receives, as one of its arguments, a pointer to the struct belonging to the next higher level of recursion, it's possible for it to duplicate that struct at the current level of recursion, which is the first thing get_best() does, via the standard memcpy() function.

At the highest level of recursion, after the memcpy(), "new_map" will point to an exact copy of "master," and "bst_ptr" will point to the first element of the best[] array.

After the current level of recursion has inherited it's game board in this way, it calls the find_moves() function.

Upon return, every available move will be indicated by a non-zero value in the corresponding element of move[] of "new_map." Unavailable moves will have zeroes in the corresponding elements.

For example, if we start with the first hole empty, the only available moves are #6 (3,1,0) and #14 (5,2,0). Therefore, "new_map.move[6]" and "new_map.- move[14] will hold values of one, and all other move[] elements will hold zeroes.

Notice how find_moves() uses triple array Indexes to determine whether or not a move is available.

The first index used is "i" from the "for" loop, which determines which of the 36 three-byte elements of the jump[] array to look at.

The address of that jump[] element is put into the variable "ptr."

Next the offset (SORC, JUMP or DEST) determines which of that element's three bytes to use.

Look at the notation "*(ptr + SORC)." This is IDENTICAL to "ptr[SORC]," since both pick up the object at the specified address. However, the "object at" method used here often generates more efficient code in the final program.

This value (ptr[SORC]), in turn, is used as an index to the map[] member of the struct.

For each move defined in the jump[] array, this function determines whether or not the current game board has a peg in the SORC hole (the peg to be moved). If not, the "if" statement is exited so the next move can be checked.

If so, it checks to see if there's a peg in the JUMP hole (the peg to be jumped over). If not, the "if" statement is exited.

If there are pegs in both of those holes, it checks to see if there's a peg in the DEST hole (the hole into which the SORC peg would go if the jump were made). Note the use of the "NOT" operator in this test. If there's a peg in the DEST hole the jump is blocked; so the "if" statement is exited.

Thus, if there are pegs in the SORC and JUMP holes, and no peg in the DEST hole, the move will be recorded in the move[] member as being available.

Otherwise, it will be left with the value zero, which each element in the move[] member was given via the standard memset() function when this function was first entered.

Understanding this requires some study, thought and going back and forth between the "find_moves()" code and the initialization of the jumps[] array, but it's important for you to fully understand the logic used in this function, and how the various array elements are accessed.

Note how the use of "ptr=jumps+i" eliminates the need to calculate the address of the proper jumps[] element for each of the three tests. The use

of this intermediate variable lets us get away with calculating that address just once instead of three times, making the routine far more efficient.

Also note how the use of the defined values SORC, JUMP and DEST make the logic clearer than it would have been had we used "0, 1 and 2."

O.K. Back in find_best(), now that all available moves have been identified, we proceed to try each and every one of them in turn.

For each available move, the first thing we do is make the move via the do_move() function. The args sent to do_move() tell it which struct ("new_map") to use, which move described in the jumps[] array ("i") to make or unmake, and a code (DO in this case) to tell it whether to make the move or reverse it.

Note how DO and UNDO allow us to use the same function to both make moves and reverse them. Obviously, this is more efficient code-wise than writing two separate functions (one to make the moves, and another to unmake them) although it does add slightly to do_move()'s execution time.

Also note how the do_move() function also makes use of the intermediate variable technique to avoid having to calculate the address of the jump[] element three separate times.

Now, back in get_best(), we call count_pegs() to get the number of pegs remaining on the board, and load the count into the variable "pegs."

Note count_pegs() returns ONE LESS than the actual number of pegs. This way, if only one peg remains (i.e., we have won the game) "pegs" will evaluate to FALSE. This makes it possible to test "pegs" with more efficient code [i.e., "if (!pegs)" is much more efficient than "if (pegs==1)"].

Now "pegs" is tested to see if a win has occurred. (Remember, this can only happen at the 13th level of recursion.) If so, the "while" loop is exited and get_best() makes a recursive return to the next higher level of recursion.

If more than one peg remains, then a recursive call is made to get_best(), with pointers to the current game board and the next best[] element as arguments.

All recursive returns will be to the next statement, namely "if (!pegs) break;". This causes the "while" loop to be exited, and an immediate return to the calling routine.

If the current move doesn't produce a win, then it's reversed via a call to do_move() with an UNDO code, and the next available move tested.

When all possibilities at this level have been exhausted without a win, then return is made to the next higher level of recursion, where the next of that level's possible moves are tried.

And so it goes until a sequence of moves is found which results in a win.

The fact that the very first statement which will be executed after each recursive return is "if (!pegs) break;" is absolutely crucial.

This code ensures that, immediately a winning sequence of moves is found (causing "pegs" to take on a value of zero) it will generate an immediate chain of recursive returns all the way back to main().

At that point main() calls the report() function to display the winning sequence of moves recorded in best[], and start the next game.

Type in, compile, and pegwins.c now.

When compiling, use MC/JCL's "o" switch. This program takes a long time to run as it is, without inefficient, unoptimized code making it even slower.

In other words, compile via the command:

```
do mc (n=moves,o,k[,t=d])
```

To run it, turn on your printer and invoke via:

```
moves >*pr
```

and go make yourself a cup of coffee or five.

Well, that about does it this time. I had hoped to get into disk I/O, but explaining pegwins.c took more space than I had anticipated. So we'll have to wait until Part 6 to learn how to handle files.