Programs To Expand Your System Capabilities

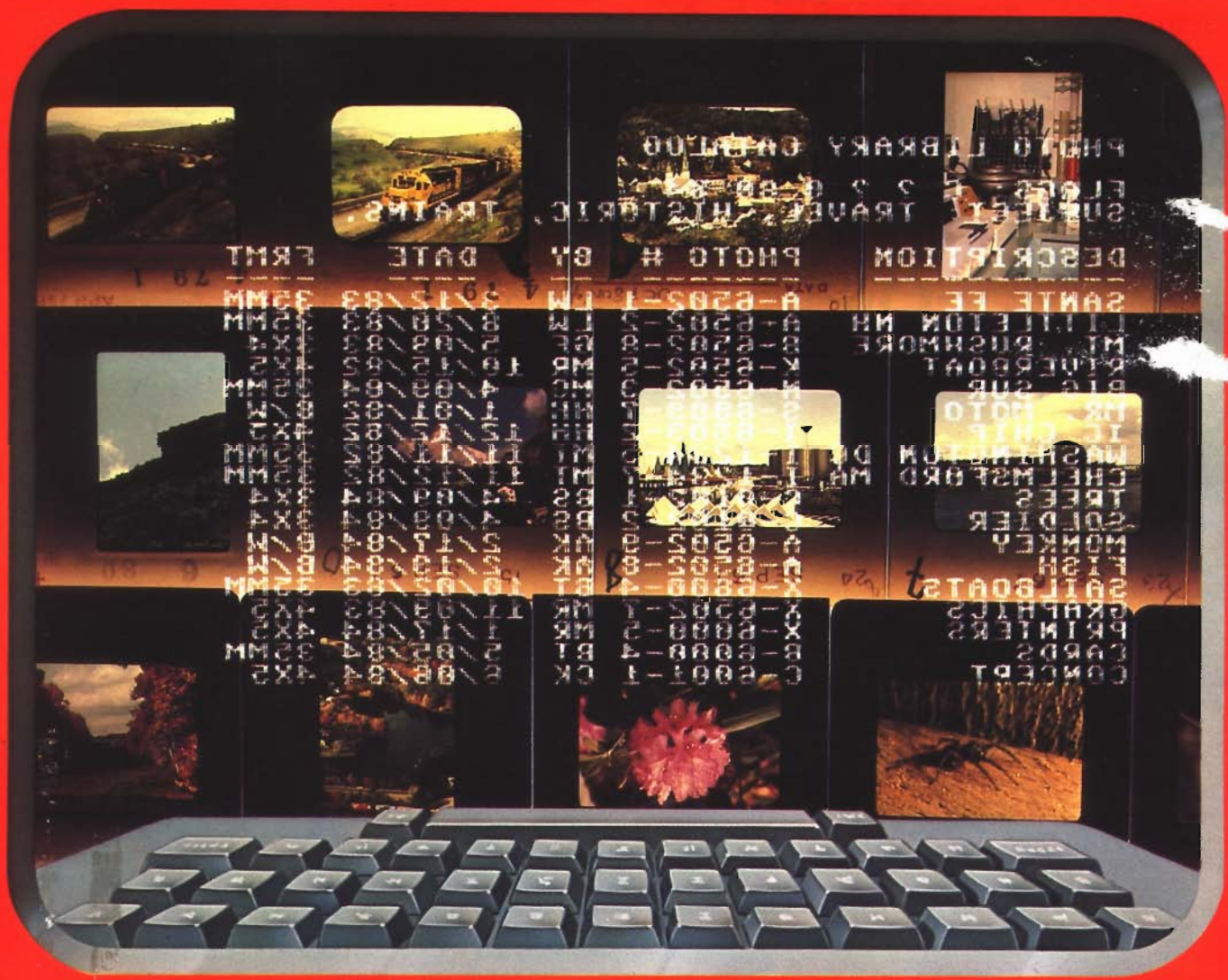
NO. 76

U.S. Edition: $2.50
International Edition: $3.50

OCTOBER 1984

MICRO™

for the Serious Computerist



Plotting Binary Trees

[redacted] for C64

[redacted] Apple

[redacted] e

68000 p-System BASIC

FORTH Interactive Input Utility

0 74470 11690 0 06

FLASH! Gets the Gold at the Computer Olympics

The Skyles Electric Works *1541 FLASH!* dashed off with the gold at the Computer Olympics here.

The *1541 FLASH!* loaded and saved programs and files *three times faster* than an unenhanced Commodore 1541 disk drive could. Faster than any other disk drive with compatible disk format.

"...faster than any other disk drive..."

Three times faster! The device delighted the home crowd, which watched the *1541 FLASH!* set a meet record, and leave its competition in the dust.

Once installed, the *1541 FLASH!* is transparent. Computer operations all remain unaffected as it speeds up every disk-related function. The *FLASH!* is a permanent installation with both a software (ROM) and a hardware component. Through keyboard commands or a hardware switch, you can even return to the old, slow loading method—if you really want to.

And there is nothing new to learn for the *FLASH!* No special tricks or techniques. Once it's in, just watch it go.

But if you're really serious about programming, the *1541 FLASH!* is a gold mine. The manual will show you how to write software allowing data transfer to and from the 1541 disk drive at speeds up to *10 times* the normal.

For programs that usually load with a " '*',8,1" command, just hit Shift/Run-Stop. A spreadsheet program like BUSICALC 3 then loads in about 25 seconds.

The *1541 FLASH!* even adds 21 extra commands for the Commodore 64 user. Some of these include editing, programming and loading commands, as well as "DOS Wedge" commands. You can ignore all these commands, though, and just enjoy the rapid disk operations.

It wowed the crowd at the Computer Olympics. Once you see its sheer speed, you'll know why. Call its coach, Skyles Electric Works, to place your order or to get more info.

***1541 FLASH!*, an add-on assembly, for the Commodore 64/1541 costs only $89.95.**

Skyles Electric Works
231E South Whisman Road
Mountain View, CA 94041
(415) 965-1735

Available from your local Commodore 64 dealer or call 1-800-227-9998.

NEW!

PRINTER ACCESSORIES FROM

DIGITAL DEVICES T.M.

U•PRINT T.M.

Expand your Atari® or Commodore® computer with Digital Devices *U • PRINT*. We make it simple to add any printer you choose. *U • PRINT* interfaces feature industry standard Centronics parallel connectors to hook up an Epson, Star, NEC, C.Itoh, Okidata, or any other printer.

- EXTRA SERIAL PORT FOR DAISY CHAINING OTHER PERIPHERALS.
- COMPATIBLE WITH ALL ATARI HARDWARE AND SOFTWARE.

U • PRINT MODEL A

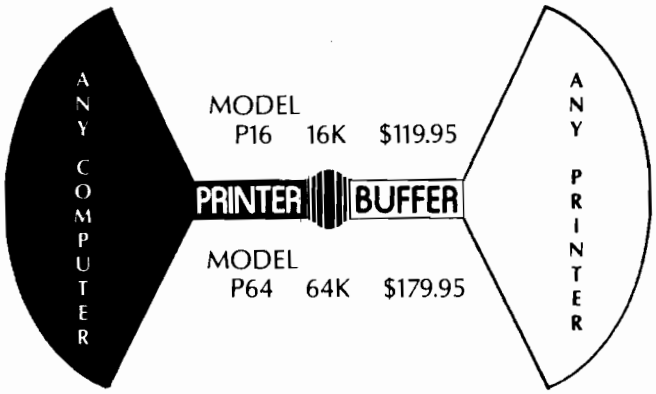
- EMULATION OF COMMODORE PRINTERS, INCLUDING GRAPHICS.
- COMPATIBLE WITH ALL COMMODORE HARDWARE AND SOFTWARE.

U • PRINT MODEL C

Compact, easy to install, and costing only $89.95, U•PRINT gives you a choice!

PRINTER BUFFER T.M.

Ever get stuck while your printer catches up? The PRINTER BUFFER eliminates waiting by rapidly accepting data in memory, then relaying it at the printer's rate, freeing the computer for your next job. User-upgradable memory (16k to 64k) allows up to 32 pages of data to be stored.



Compatible with U•PRINT and other industry-standard hardware, the PRINTER BUFFER is the low-cost way to make your computer even more productive!

Quality Products Made In USA From

DIGITAL DEVICES Corporation

430 Tenth Street, Suite N205 Atlanta, Georgia 30318
In Georgia (404) 872-4430; Outside Georgia (800) 554-4898

From the editors of
A.N.A.L.O.G. Computing

$14.95

THE A.N.A.L.O.G. COMPENDIUM

The best ATARI® Home Computer Programs from the first ten issues of A.N.A.L.O.G. Computing Magazine.

All Compendium programs are available on DISK.

The **ANALOG Compendium** is available at selected book and computer stores, or you can order it direct. Send a check or money order for $14.95 + $2 shipping and handling to: **ANALOG Compendium, P. O. Box 615, Holmes, PA 19043.**

Or you can order by phone with MasterCard or VISA. Call toll free: 1-800-345-8112 (in PA, call 1-800-662-2444). **For orders outside the U.S., add an additional $5 air mail, $2 surface.**

highlights

Fast Bit Map Plotting. This is the first in a multi-part series that will discuss the theory of plotting hi-resolution points, lines and other picture elements in the Commodore 64 — and provide a collection of assembly level subroutines to perform these functions. The subroutines, which may be called from BASIC, provide a very fast and efficient method of 'unlocking' the hi-resolution capabilities of the C64. The second article in the series will add the routines necessary to draw lines between points. These will allow the C64 programmer to generate USR calls that are equivalent to the Applesoft HPLOT routines, and will let the C64 user convert programs written for the Apple, such as 'Plotting Binary Trees' in this issue, to the C64.

Plotting Binary Trees — Binary trees are a form of mathematical graph that display interesting properties. The short program provided calculates all of the information required to plot these binary trees on a microcomputer display. The user may specify the parameters that govern the 'growth' of the tree and observe the results in a very graphic fashion. While the plotting portion of the program is specific to Applesoft, relying on the HPLOT function, it should be convertable to almost any other micro.

Database Management Systems. Approximately 42% of the MICRO readers reported that they use their systems for database management. This article explores the significant features of database managers (DBMs) and can be used as a guide to selecting the appropriate package for your applications on your computer. Part 1 of the two part article looks at the DBM features in general and is applicable to all microcomputers. Part 2, scheduled to appear in next month's issue, applies the concepts developed in Part 1 to evaluate a large number of the DBMs available for the Commodore 64.

BASIC/ML Data Transfer. Many computer problems are best solved by combining the ease of BASIC with the speed of machine language programming. Unfortunately, BASIC is not as supportive of ML data as it might be. Sure, you can PEEK and POKE all day, but then you are apt to lose all of the efficiency and speed you set out to achieve. Four techniques are presented to permit BASIC and ML data to work together.

A Very Moving Message. Sometimes an effect that looks simple can actually be the result of very complex operations. This article includes a program that allows a message to be scrolled across the screen of the Commodore 64 while other activities are going on. Simple? Not really. As the article shows, it requires a use of interrupts, split-screen capability and smooth scrolling to make it work. Each of these concepts is explained and the resulting demonstration program makes them clear. The 'Moving Message' routines can be added to your own programs to make them look attractive, professional and make them easier to use.

Publisher/Editor-in-Chief
Robert M. Tripp

Associate Publisher
Cindy Kocher

Production Manager
Jennifer Collins

Technical Editor
Mark S. Morano

Technical Editor
Mike Rowe

Advertising Manager
William G. York

Circulation Manager
Linda Hensdill

Office Manager
Pauline Giard

Shipping Director
Marie Ann Wessinger

Comptroller
Donna M. Tripp

Accounting
Louise Ryan

Contributing Editors
Cornelis Bongers
Phil Daley
David Malmberg
John Steiner
Jim Strasma
Paul Swanson
Richard C. Vile, Jr.
Loren Wright

MICRO is published monthly by:

MICRO
P.O. Box 6502
Chelmsford, MA 01824

Second Class postage paid at:
Chelmsford, MA 01824 and additional mailing offices.
USPS Publication Number: 483470.
ISSN: 0271-9002.
Send subscriptions, change of address, USPS Form 3579, requests for back issues and all othe fulfillment questions to:
MICRO
P.O.Box 6502
Chelmsford, MA 01824
or call 617/256-3649.

Subscription Rates: (per year):
U.S. $24.00 or $42.00 for two years
Foreign surface mail: $27.00
Air mail: Europe $42.00
Mexico, Central America, Middle East, North Africa, Central Africa $48.00
South America, South Africa, Far East, Australia, New Zealand $72.00

MICRO™

for the Serious Computerist

OCTOBER 1984

Departments

editorial

Autumn is my season for reflection.

Is it possible that MICRO started publication seven years ago and is now starting its eighth year? Have we changed much since that first issue in October 1977? Have we accomplished our original goals? Wondering, I went back to the first issue and reread my editorial. I think that you might enjoy reading it too, in its original published form.

Is the 6502 still number one? With the popularity of the Apple II, Commodore 64, VIC-20, Atari and other 6502-based systems, there is probably no argument now, but that was not the case back then. Although newer chips are making their mark, the 6502 continues to be the leader.

Have we attracted "individuals who are industrious, able, cooperative, adventurous and communicative" as readers? Examine the results of the June 1984 survey and judge for yourself.

If MICRO is the "most useful journal" for you, then "We're Still Number One!"

Robert M. Tripp

On The Cover

```
PHOTO LIBRARY CATALOG

FLAGS: C 7 7 0 78 81
SUBJECT: TRAVEL, HISTORIC, TRAINS.

DESCRIPTION     PHOTO #     BY    DATE      FRMT
-----------     -------     --    ----      ----
SANTE FE        A-6502-1    LW   3/12/83    35MM
LITTLETON NH    A-6502-2    LW   8/20/83    35MM
MT. RUSHMORE    B-6502-8    GF    5/09/83   3X4
RIVERBOAT       K-6502-5    MR  10/15/82    4X5
BIG SUR         M-6502-9    MG    4/09/84   35MM
MR. MOTO        S-6809-7    HH    1/01/82   B/W
IC CHIP         I-6809-2    HH   12/12/82   4X5
WASHINGTON DC   I-1234-5    MT   11/11/82   35MM
CHELMSFORD MA   I-1234-7    MT   11/12/82   35MM
TREES           C-0102-1    BS    4/09/84   3X4
SOLDIER         C-0102-2    BS    4/09/84   3X4
MONKEY          A-6502-9    AK    2/17/84   B/W
FISH            A-6502-8    AK    2/19/84   B/W
```

When someone mentions 'Data Base Management', most of us probably think in terms of business computer-oriented materials — mailing lists, inventory control and so forth. DBM's can be used for many personal uses as well. The cover shows a collection of photographic slides that can be encoded and selected via a DBM. Other personal examples could include large record and tape collections, hobby classification systems and more.

We're Number One !

An Editorial

We're number one in microcomputer systems. With over twelve thousand KIM-1 microcomputers in the filed and a thousand per month being ordered, plus a good number of Apple I and Apple II systems, plus a variety of OSI units, Plus the Jolts, Data Handlers, and other 6502-based systems, plus the huge numbers of PETs and Microminds that have been ordered, plus a lot of home-brew 6502 systems - it all adds up to a tremendous number of 6502-based microcomputer systems in use throughout the world. Adding to this number are the one and one-half million 650x chips purchased by Atari for some of their games. We've come a long way in the past year.

We're number one in microprocessor power. Microchess for the KIM-1 took 1.1K and for the 8080A took about 2.5K. Of thirty-one BASICs tested and reported in Kilobaud, the four 6502 versions placed in the top five spots, yielding only second place to the Z-80 running at 4 MHz. The 6502's many addressing modes make it very efficient and easy to program.

We're number one in user participation. Maybe there is some process of "natural selection" which attracts individuals who are industrious, able, cooperative, adventurous and communicative to the 6502. While users of other microprocessor chips have been "spoonfed" via company supported user notes and user libraries, the 6502 users have been "doing their own thing" as evidenced by the activity level of many local 6502 groups and the success of the KIM-1/6502 User Notes.

We're number one since this is our first issue. We would like to really become the most useful journal in the whole microcomputer field, not the largest, just the best. We are undertaking the venture with the conviction that there is a need for a journal to help bring all of the separate parts of the 6502 world together and with the belief that 6502 users will each do what they can to support the effort.

Plotting Binary Trees

by Luther K. Branting
Denver, Colorado

A Program to Plot an Interesting Class of Graphs: Binary Trees

Trees are a form of graph that play a special role in computer science. Most artificial intelligence is based upon tree-like decision paths and trees are used to model such diverse natural phenomena as river basins, languages and plant growth. Graphs of trees also form an endless variety of fascinating and beautiful patterns.

A tree is a graph consisting of a vertex called a root and one or more line segments branching from the root. Additional branchings may occur from the end points of each branch. A binary tree is a special form of tree in which the root has exactly two branches and each of the branches, in turn, has exactly two more branches. With each additional branching, or generation, the number of end points doubles.

Figure 1 shows a two-generation binary tree. A and B are the first generation branches. C, D, E and F are second-generation branches. In this tree, each left branch is inclined at an identical angle, LA, from the previous branch. Similarly, RA is the angle of each right branch. All the branches are the same length.

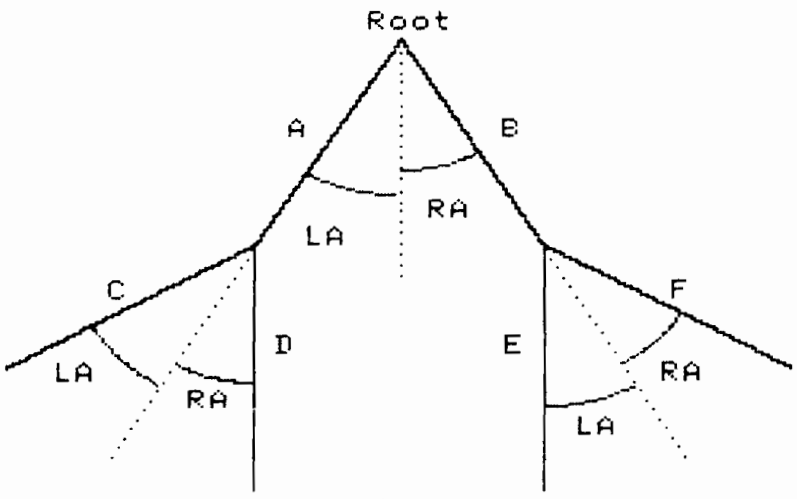


Figure 1

Figure 2 shows the same tree except that now each left branch is only .6 times as long as the previous branch.

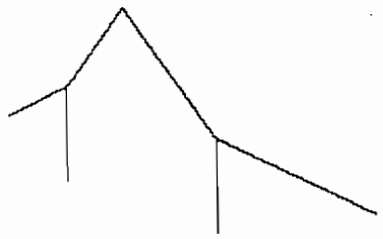


Figure 2

The ratio of each branch to the previous branch is called the growth factor of the branch. In general, each tree of this type can be specified by giving the left and right branch angles, the left and right growth factors, the length of the first two branches and the number of times the branching is to occur.

The program listed below prompts the user for the number of generations of the tree, left and right branch angles and growth factors and the scale, a number which is multiplied by the left and right growth factors to obtain the lengths of the initial left and right branches. After the tree is drawn, pressing any key will clear the screen and present the user with the options of plotting a new tree, sending the tree just plotted out to a printer, or quitting.

As each new generation is drawn, the arrays X1(i) and Y1(i) hold the X and Y coordinates of the end point of the ith branch of the existing tree. A(i) holds the angle of the ith branch and L(i) holds its length. Similarly, X2, Y2, A2 and L2 contain the coordinates, angle and length of each new branch being drawn. After each generation is drawn, the values in X2, Y2, A2 and L2 are transferred to X1, Y1, A1 and L1 so that the end points of the current branches can be used as the starting points of the new generation of branches.

When starting out, it is best to use growth factors between 1 and 0.5. Angles that are the result of dividing 360 by an integer, like 30, 45, 60 and 72 degrees, seem to produce the most attractive trees. There seems to be an affinity between angles that are a multiple of 30 degrees and the growth factors .618 and 1, and between angles that are a multiple of 45 degrees and the growth factors .707 and 1. In the interest of speed and simplicity, the program does not proportion the tree being drawn so that it fits within the screen. It is up to the user to select an appropriate scale. If the tree is microscopic, try a larger scale. If the tree falls outside of the screen, the program prompts for a smaller scale.

By plotting a series of trees that vary in their angles and growth factors only slightly, one can create the illusion of flowers opening or crystals growing. Some examples of trees are shown below. Note that in Figure 3, the end points of the tree are converging on a

fractal curve similar to the Koch curve described in Plotting Fractals on Your Computer, Micro No. 70, March 1984.

The Program

The calculation portion of this program will work in almost any BASIC. The plotting assumes the Apple II routines to HOME (clear screen and home cursor), VTAB and HTAB (to position the cursor) and HPLOT (to plot a line between two points). If your BASIC does not have these routines, you must supply them. Commodore 64 owners: Loren Wright's *Fast Bit Map Plotting*, appearing in this issue, and *Fast Line Plotting*, appearing in next issue, will provide the necessary routines to implement the Binary Tree Plotting.

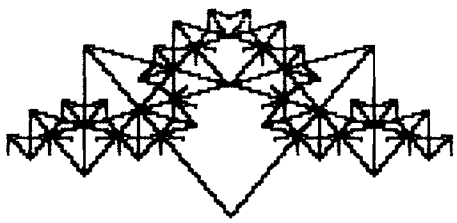
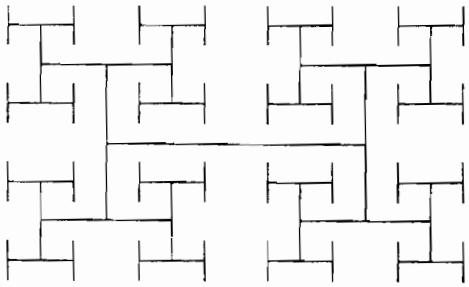
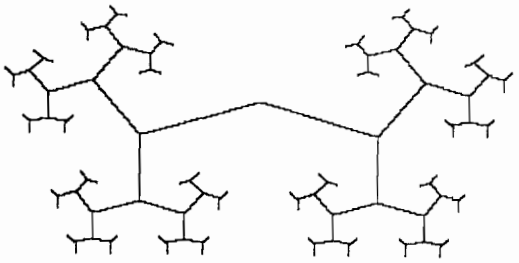


Figure 3

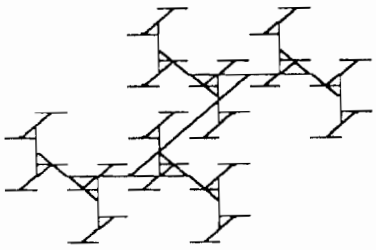
```
  1  REM  *** BINARY TREE PLOTTING   PROGRAM
  5  ONERR  GOTO 600:
     REM   ***  IF TREE IS TOO LARGE TO FIT ON THE
     SCREEN, PROMPT FOR A LARGER SCALE
 10  DIM X1(128): DIM Y1(128): DIM X2(128):
     DIM Y2(128)
 20  DIM A1(128): DIM A2(128): DIM L1(128):
     DIM L2(128)
 30  HOME : INPUT "GENERATIONS?(RANGE1-7) ";G
 40  IF G > 7 OR G < 1 THEN  GOTO 30
 50  INPUT "LEFT ANGLE? ";LA
 60  INPUT "RIGHT ANGLE? ";RA
 69  REM  *** CONVERT FROM DEGREES TO RADIANS
 70  LA = LA * 3.14159266 / 180:
     RA = RA * 3.14159266 / 180
 80  INPUT "LEFT GROWTH FACTOR? ";LGF
 90  INPUT "RIGHT GROWTH FACTOR? ";RGF
100  INPUT "SCALE? ";SCL
110  HGR2 : REM   *** CLEAR GRAPHICS SCREEN
119  REM  *** INITIALIZE STARTING POINT TO SLIGHTLY
     ABOVE THE CENTER OF THE SCREEN
120  Y1(0) = 86:X1(0) = 140
140  L1(0) = SCL:A1(0) = 0
150  SIZE = 1
160  FOR A = 1 TO G
170  SIZE = SIZE * 2
171  REM  ***  SIZE IS THE NUMBER OF  END POINTS OF
     THE GENERATION BEING DRAWN
180  FOR I = 0 TO SIZE / 2 - 1
185  RI = 2 * I:LI = 2 * I + 1
186  REM  *** RI IS THE INDEX OF THE RIGHT BRANCH;
     LI IS THE INDEX OF THE LEFT
189  REM   *** CALCULATE ANGLES OF RIGHT AND LEFT
     BRANCHES
190  A2(RI) = A1(I) + RA:A2(LI) = A1(I) - LA
199  REM   *** CALCULATE LENGTH OF RIGHT AND LEFT
     BRANCHES
200  L2(RI) = L1(I) * RGF:L2(LI) = L1(I) * LGF
209  REM  *** CALCULATE X AND Y COORDINATES OF END
     POINT OF RIGHT BRANCH
210  X2(RI) = X1(I) +  SIN (A2(RI)) * L2(RI)
220  Y2(RI) = Y1(I) +  COS (A2(RI)) * L2(RI)
229  REM  *** PLOT RIGHT BRANCH
230  HPLOT X1(I),Y1(I) TO X2(RI),Y2(RI)
239  REM   *** CALCULATE X AND Y COORDIATES OF END
     POINT OF LEFT BRANCH
240  X2(LI) = X1(I) +  SIN (A2(LI)) * L2(LI)
250  Y2(LI) = Y1(I) +  COS (A2(LI)) * L2(LI)
259  REM   ***  PLOT LEFT BRANCH
260  HPLOT X1(I),Y1(I) TO X2(LI),Y2(LI)
270  NEXT I
279  REM  *** SHIFT ARRAY VALUES FOR CALCULATION OF
     NEXT GENERATION
280  FOR I = 0 TO SIZE - 1
290  X1(I) = X2(I):Y1(I) = Y2(I)
300  L1(I) = L2(I):A1(I) = A2(I)
310  NEXT I
320  NEXT A
329  REM  *** WAIT FOR A KEY TO BE PRESSED,
     THEN CLEAR SCREEN
330  GET A$: TEXT : HOME
340  PRINT "INPUT:"
350  VTAB 6: HTAB 10: PRINT "Q—TO QUIT"
360  VTAB 10: HTAB 10: PRINT "P—TO PRINT TREE"
365  VTAB 14: HTAB 10: PRINT "N—FOR NEW TREE"
370  GET A$: IF A$ = "P" THEN  GOTO 400
375  IF A$ = "N" THEN  HOME : GOTO 30
380  IF A$ <  > "Q" THEN  GOTO 330
390  END
399  REM  ***  PROVIDE APPROPRIATE COMMAND FOR YOUR
     PRINTER
400  PR# 1: PRINT  CHR$ (9);"G2DL"
410  PRINT "LEFT ANGLE= ";LA * 180 / 3.14159266
420  PRINT "RIGHT ANGLE= ";RA * 180 / 3.14159266
430  PRINT "LEFT GROWTH FACTOR= ";LGF
440  PRINT "RIGHT GROWTH FACTOR= ";RGF
450  PRINT "SCALE= ";SCL
460  PR# 0
470  HOME : GOTO 340
600  TEXT : HOME :
     PRINT "TREE PARTIALLY OFF SCREEN.":
     PRINT "TRY AGAIN USING SMALLER SCALE":
     GOTO 100
```



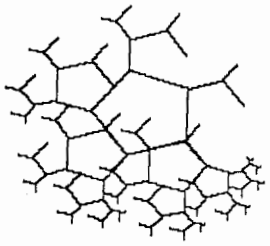
LEFT ANGLE= 90
RIGHT ANGLE= 90
LEFT GROWTH FACTOR= .707
RIGHT GROWTH FACTOR= .707
SCALE= 90



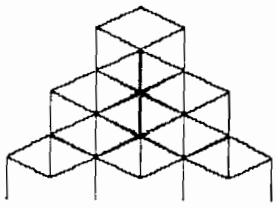
LEFT ANGLE= 72
RIGHT ANGLE= 72
LEFT GROWTH FACTOR= .618
RIGHT GROWTH FACTOR= .618
SCALE= 100



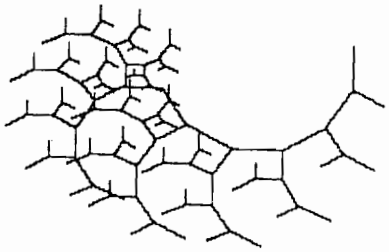
LEFT ANGLE= 45
RIGHT ANGLE= 135
LEFT GROWTH FACTOR= .707
RIGHT GROWTH FACTOR= .707
SCALE= 60



LEFT ANGLE= 72
RIGHT ANGLE= 36
LEFT GROWTH FACTOR= 1
RIGHT GROWTH FACTOR= .618
SCALE= 30



LEFT ANGLE= 120
RIGHT ANGLE= 120
LEFT GROWTH FACTOR= 1
RIGHT GROWTH FACTOR= 1
SCALE= 25



LEFT ANGLE= 90
RIGHT ANGLE= 30
LEFT GROWTH FACTOR= .707
RIGHT GROWTH FACTOR= 1
SCALE= 25

Additional Examples of Binary Tree Plotting

Portrait of a Serious Computerist
Results of the June 1984 MICRO Survey

What is your age?

33%	AGE 30–39
25	AGE 40–49
20	AGE 20–29
14	AGE 50–59
5	AGE 60+
4	AGE -19

What is your occupation?

19%	Engineer
19	Other
17	Professor/teacher
16	Programmer/Analyst
11	Technician
6	Student
6	Self Employed
3	Lawyer
2	Doctor
1	Business person

What is your formal educational level?

34%	Bachelor's degree
33	Advanced degree
16	High school graduate
14	Associate degree
2	Fewer than 12 years
2	Para-professional degree

What is your annual household income before taxes?

26%	$30,000–39,999
24	$50,000+
19	$40,000–49,999
17	$20,000–29,999
11	Less than $20,000

What microcomputer(s) do you use?

43%	Apple II
38	Other
37	Commodore 64
14	Atari
11	VIC
10	PET/CBM
8	OSI
6	AIM
6	KIM
5	Macintosh
5	SYM
4	TRS-80 Color Computer
4	Other 6502
4	Other 6809

Where do you use the above computers?

94%	Home
51	Work
11	School
3	Other

Approximately how much have you spent on your computer hardware so far?

22%	$1000–1999
19	$2000–2999
16	$3000–3999
13	$5000–9999
12	$4000–4999
7	$500–999
7	$10,000+
2	-$500

Approximately how much do you plan to spend on your computer hardware in the next year?

34%	$500–999
27	-$500
23	$1000–1999
6	$2000–2999
2	$3000–3999
2	$4000–4999
2	$5000–9999
2	$10,000+

What additions have you made to your basic system?

82%	Disk drives
81	Printer
52	Modem
51	Parallel interface
41	RAM cards
34	Serial interface
19	Z80 card
15	Graphics Tablet
4	Hard disk
3	68000 card
2	6809 card

What additional hardware changes or upgrades do you plan to make to your system?

26%	Disk drives
25	Modem
23	Printer
13	68000 card
13	Hard disk
13	Graphics Tablet
10	RAM cards
7	Z80 card
5	Serial interface
4	Parallel interface
1	6809 card

Have you ever constructed a computer, computer board, or major computer equipment?

56%	No
41	Yes

Have you switched from one computer to another?

53%	No
43	Yes

Approximately how much have you spent on your computer software so far?

30%	$500-999
28	$200-499
15	-$200
14	$1000-1999
12	$2000+

Approximately how much do you expect to spend on computer software in the next year?

39%	$200-499
34	-$200
17	$500-999
5	$1000-1999
3	$2000+

How do you use your computer equipment?

74%	Word processing
66	Hobby
57	Software development
52	Entertainment
49	Business
42	Database management
40	Educations
35	Telecommunications
35	Graphics
19	Hardware development
11	Other

What languages do you use?

96%	BASIC
72	6502 Assembler
30	Pascal
26	Forth
16	Fortran
15	Other
13	LOGO
11	C
9	68000 Assembler
7	COBOL
5	6809 Assembler
3	APL
3	LISP

If you write programs, what type of programming do you spend most of your time developing?

42%	Software development utilities
37	Other
29	Business applications
7	Games

How would you rate your present microcomputer knowledge:

	Software:
51%	Intermediate
44	Advanced
5	Elementary
	Hardware:
53%	Intermediate
28	Advanced
18	Elementary

Is MICRO:

55%	Just right
31	Not technical enough
4	Too technical

A Few Notes

The 1984 MICRO Survey Form was printed as a self-mailer in the June 1984 issue. The results were converted to computer-readable form using an Apple II and an Apple Graphics Tablet. We have presented here the results that we felt would be of interest to our readers. We have sorted each question so that the answers are ranked in descending order to make the results easier to follow. All figures are a percentage of the total responses to each question. Since the results are rounded, they may not always equal exactly 100%. Some may total less than 100% if some readers did not answer the question. Other questions, in which readers might make several choices, will add up to over 100%. For example, "What microcomputer(s) do you use?" responses total 195%, indicating that MICRO readers use, on the average, two (2) microcomputers.

Some Significant Results

Some of the results that I find particularly significant in characterizing the MICRO reader are:

- age (76% are 30 or older)
- education level (75% have a Bachelor or Advanced degree)
- use of microcomputers at work (51%) as well as at home (94%)
- amount invested (48% have spent over $3000)
- programming knowledge (95% rate themselves as intermediate or advanced)
- programming languages (72% use 6502 assembler, 30% Pascal, 26% FORTH)
- hardware skills (41% have built major computer projects, 81% rate themselves intermediate or advanced)

This is a pretty heavy group!

Conclusions

We are pleased to see that our readership is so qualified, that they really are, for the most part, "Serious Computerists". For the 55% of you that think MICRO is technically "just right", we are glad that you are satisified. For the 31% reporting that we are "not technical enough", wait until you see what we have in store for you in upcoming issues! Some sophisticated theory and programs for adding shading to graphics. A 'build-it-from-scratch' 68000 co-processor to work with your Apple or Commodore 64, complete with a 68000 monitor and a 6502 cross assembler. And, if you are into mathematical applications, a very high-level series on fast equation solving. And for the 4% who find us "too technical", well, there are literally hundreds of other computer magazines out there ready to serve your needs. MICRO will continue to strive to serve those of you who are really serious about microcomputing.

Fast Bit Map Plotting
for the Commodore 64

by Loren W. Wright
Dracut, Massachusetts

The Commodore 64 has great high resolution color capabilities built-in. Assembly level routines are presented to support fast bit map plotting and to provide the basis for a hi-res support package.

Introduction

The Commodore 64 has a very capable system of bit map graphics. In the high-resolution mode two or more colors are available, and the resolution is 320 dots across by 200 vertically. In the multicolor mode four or more colors are available, with the horizontal resolution reduced to 160 dots. However, access to bit-map graphics from BASIC is poor, requiring a series of cryptic POKEs, PEEKs, ANDs, and ORs, instead of PLOT, GRAPHICS, and COLOR commands. Even worse, BASIC is very slow at performing the necessary tasks. Clearing the bit map (8,000 bytes) takes 30 seconds, and even changing one of the plot colors takes several seconds. I presented simple machine-language routines for these tasks in a Commodore Compass article (MICRO 68:43, Jan 1984) and these routines have been reassembled to work with XYPLOT and BMCALC.

Another area where a machine-language program can make a big difference is in the actual plotting of points on a bit map screen. In this article I present a routine to calculate the appropriate byte in bit-map memory, given the x- and y-coordinates of the desired point. First, for those who want to "load-and-run," I provide a sample driver routine that uses the values of the BASIC variables X, Y, and C to plot, erase, or toggle points on the HiRes bit-map screen. Those who would like to see a demonstration of some simple machine-language arithmetic will want to read the detailed discussion of the BMCALC routine later in this article.

The calculation routine may also be used to convert sprite positions to use a sprite as a pointer or pen. Other uses include converting sprites or characters to bit-map images, and vice versa. Routines for these applications may be the subjects of future articles.

The XYPLOT Routine

XYPLOT works equally well with HiRes and multicolor. Once you have executed INIT the points will be plotted automatically in the current mode.

Using XYPLOT

Be sure you have XYPLOT properly installed, either with a direct memory load or with BASIC READ/DATA routine. After your bit map mode is in effect, you must initialize once with SYS 49216. The INIT routine sets up the proper data for both the clear routines and BMCALC. Then all you need to do is set the BASIC variable X within the allowable range of 0 to 319 (or 0 to 159 for multicolor), the variable Y between 0 and 199, and variable C to 0, 1, or 2 (0, 1, 2, 3, or 4 for multicolor). Then SYS 49219. That's it!

HiRes:

C0 plots a point in background color.
C1 plots a point in foreground color.
C2 toggles the point, i.e., a background point becomes a foreground point, and vice versa.

Multicolor:

C0 plots a point in background color (53281).
C1 plots a point in Color 1.
C2 plots a point in Color 2.
C3 plots a point in Color 3.
C4 toggles the point: 0 becomes 3, 1 becomes 2, 2 becomes 1, and 3 becomes 0.

Any other value for C causes nothing to happen. By the way, this is the same plot-type useage as SIMON's BASIC.

A sample plotting program:

```
WW=49152
SYS WW+64
FOR X=Ø TO 199
Y=X
SYS WW+67
NEXT X
```

This makes a straight diagonal line up from the lower left corner. The routine assumes a lower left orgin. Failure to perform INIT, or performing it at the wrong time, is the only possibly fatal error.

How it Works

The routine must perform the following tasks:

1. Determine the values of BASIC variables X, Y, and C.

2. Invert Y by subtracting it from 199.

3. Set up BMCALC and execute it.

4. Read the contents of the calculated byte and modify it according to the value of C.

The subroutine VARSET uses three C-64 ROM routines to 1) find where the value of a variable is stored, 2) load it into the floating-point accumulator, and 3) convert it into an integer. The floating-point variables X, Y, and C are set up by storing the ASCII of the appropriate letter into $45 and $00 into $46.

Y is inverted by subtracting it from 199. Since the values of X increase from left to right, while the bits increase from right to left within a bit-map byte, the bit position in HiRes mode is calculated by first ANDing with 7, then EORing with 7.

For HiRes mode, a table HRTBL of eight bit masks is used to calculate the new value to store back into bit-map memory. The bit position is used as an index into the table.

Setting a point to foreground color means ORing the table value. Setting a point to background color means EORing the table value with 255 and ANDing the result with the bit-map byte. Toggling a point is simply a matter of EORing with the table value.

In multicolor mode, the proper bit pair is calculated by ANDing and then EORing the X-coordinate with 3. The BMCALC routine takes X values in the range 0 to 319, so the multicolor X value gets temporarily multiplied by 2 while BMCALC is using it. A table of four bit-pair masks is used to save the contents of the other three bit pairs in the byte. In the toggle mode, another table is used to read the current contents of the bit pair so that it can be inverted.

The BMCALC Routine

The Problem

For most plotting on the bit-map screen, it is convenient to use it as a big sheet of graph paper with x-coordinates running from 0 on the left to 319 on the right, and y-coordinates running from 0 at the bottom to 199 at the top. However, bit-map memory is not organized that way. Instead, it is organized as if the memory were character definitions. Each byte codes for a row of eight pixels.

```
BYTE000 BYTE008 ... BYTE312
BYTE001 BYTE009 ... BYTE313
BYTE002 BYTE010 ... BYTE314
BYTE003 BYTE011 ... BYTE315
BYTE004 BYTE012 ... BYTE316
BYTE005 BYTE013 ... BYTE317
BYTE006 BYTE014 ... BYTE318
BYTE007 BYTE015 ... BYTE319

BYTE320 BYTE328 ... BYTE632
   .       .          .
   .       .          .
   .       .          .
```

Having bit-map memory organized this way is convenient for setting colors, since screen memory (which normally *does* hold characters) is used for the colors. It is not very convenient for plotting points, though. Following are the calculations required in BASIC to convert an x,y point to the appropriate byte and bit in HiRes bit-map memory:

```
BMLOC=start of bit-map memory
ROW=INT(Y/8)
COLUMN=INT(X/8)
LINE=Y AND 7
BIT=7-(X AND 7)
BYTE=BMLOC+ROW*320+
  COLUMN*8+LINE
```

To set a pixel to the foreground color:

```
POKE BYTE,PEEK(BYTE) OR 2↑BIT
```

To set a pixel to the background color:

```
POKE BYTE,PEEK(BYTE) AND
 (255-2↑BIT)
```

No wonder plotting a point takes so long! The machine-language routine does exactly the same thing, only much faster. In describing the program, I will use the same terminology as above.

ROW and COLUMN describe the character position of the point. ROW can have values from 0 to 24, and COLUMN can have values from 0 to 39. There are 1000 different row and column combinations, just like a character screen. As well as helping in calculating the value of BYTE, these can be used to calculate the appropriate bytes in character and color memory for setting and changing colors.

LINE describes the position (0 to 7) within the "character." For instance, byte 322 in the diagram above has a LINE value of 2.

Putting the Address Together

Each address consists of 16 bits, or two 8-bit bytes. The machine language program puts the address together from different sources.

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

These two bits are determined by the bank being used by the 6567 (VIC II). The bank is controlled by bits 0 & 1 of port A of one of the 6526's (CIA). However, these bits are inverted (3 indicates bank 0, 2 indicates bank 1, etc.), so there are two steps involved in the calculation:

1. Invert the two bits.

2. Get them from positions 0 & 1 to positions 14 & 15 of the address, or positions 6 & 7 of its high byte.

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

This bit is determined by the location of the bit map within the bank. If the bit map is in the lower 8K of the 16K bank, then bit 13 must be 0. If it is in the upper 8K, then the bit must be 1. Bit 3 of register $18 in the 6567 (address $D018 or 53272) controls this.

The INIT routine performs the calculation of bits 13, 14, & 15. The bank and bit-map location are never changed in the middle of a plotting session, so some speed can be gained by separating these calculations and performing them once at the beginning. INIT must be performed while the bit-map screen is in effect, though. Performing INIT while in normal character mode will likely result in points getting plotted in page 0, the stack, and your BASIC program! The result is stored in BMLOC, and this value is ORed into the high byte BMPTR1 near the end of the main calculation routine. In addition, INIT calculates the start of screen memory and saves it for use by the clearing routines.

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

Calculating these bits occupies most of the BMCALC routine. The following expression represents what we want to end up with in these bits:

```
320*ROW + 8*COLUMN
```

Multiplication and division in machine language are not the easiest things, but it helps when one of the numbers involved is a power of two. Then all you need to do is shift the other number left to multiply and right to divide. For instance, to multiply by 64, just shift the

other number six places to the left (64=2↑6). A 16-bit multiplication works automatically, if you shift the low byte, immediately followed by shift of the high byte. The high byte must be shifted using a 'rotate' instruction, so that the carry will transfer the bit pushed out of the low byte.

ASL of low byte:

```
C < 7 < 6 < 5 < 4 < 3-
  < 2 < 1 < Ø < Ø
```

ROL of high byte:

```
C < 7 < 6 < 5 < 4 < 3-
  < 2 < 1 < Ø < C
```

(Carry from low byte)

You may have noticed that 320 is not a power of two. However, it does equal 256 + 64, so that simplifies things.

Before we perform the above calculation, though, we must have values for ROW and COLUMN. ROW is INT(YPOS/8) and COLUMN is INT(XPOS/8). All we have to do is shift each number three bits to the right. The INT operation occurs automatically when the right three bits fall off without being saved! The only complication is that XPOS is contained in two bytes. By first shifting the accumulator, which starts with the value of XHI, followed by COLUMN, which starts with the value of XPOS, that 9th bit is automatically shifted into the low byte. (This time the rotate instruction is used on the low byte, so that it will pick up the carry, containing the bit pushed from the high byte.) For convenience, ROW, which starts with the value of YPOS, is shifted at the same time. COLUMN and ROW end up with the correct values for color calculations.

```
  Acc                COLUMN
 (value of XHI)      (val of XPOS)
Ø Ø Ø Ø Ø Ø Ø X    X X X X X X X X
```

ROW (value of YPOS)

```
Y Y Y Y Y Y Y Y
```

Result after 3 shifts:

```
Acc (discarded)    COLUMN
Ø Ø Ø Ø Ø Ø Ø Ø     Ø Ø X X X X X X
```

ROW

```
Ø Ø Ø Y Y Y Y Y
```

If we rewrite the expression for calculating bits 3-12, it becomes:

```
256*ROW + 64*ROW + 8*COLUMN
```

The first and third parts of the calculation are trivial. To get 256*ROW, all we have to do is add it to the high byte (BMPTR + 1), rather than the low byte (BMPTR) of the address. To get 8*COLUMN take XPOS and remove the three low bits. COLUMN was calculated by dividing XPOS by 8, so the only difference between XPOS and 8*COLUMN is the three lost bits.

64*COLUMN is only difficult because it involves shifting across two bytes. As I explained above, you shift the low byte first. The bit pushed off the left end goes into the carry. If you then perform a rotate on the high byte, the carry is shifted into bit 0 of the high byte. Six successive shift and rotate sequences results in a multiplication by 64.

```
BMPTR+1 (starts=Ø) Acc (val of ROW)
Ø Ø Ø Ø Ø Ø Ø Ø    Ø Ø Ø Y Y Y Y Y
```

After 6 shift & rotate sequences:

```
BMPTR+1            Accumulator
Ø Ø Ø Ø Ø Y Y Y     Y Y Ø Ø Ø Ø Ø Ø
```

(Sharp readers may have noticed that I could have accomplished the same thing with only two shifts in the opposite direction, but that's a little confusing. 64*ROW256*ROW/4)

All that's left is putting the pieces together. This is accomplished by adding up the components. Then BMLOC is ORed into BMPTR + 1.

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

The final three bits are LINE, the byte (0-7) within the "character". This is calculated by ANDing 7 with YPOS. LINE is ORed into BMPTR.

Applying BMCALC

The simplest way to use BMCALC from BASIC (without a machine language driver such as XYPLOT) is outlined below. The bit map should already be set up and protected, and probably cleared.

1. Perform INIT with SYS 49216 as soon as the bit-map mode is entered, but not before. This needs to be done only once at the beginning of the program, unless you change banks or bit-map locations.

2.
```
POKE 49168,X AND 255:
   POKE 49169,-(X> 255):
   POKE 49170,Y
```

3. SYS 49222

4. BY=PEEK(251)+256*PEEK(252)

5.
```
POKE BY,PEEK(BY) OR
  2↑(7-(X AND 7))
```
to plot a foreground point, or

```
POKE BY,PEEK(BY) AND
  255-2↑(7-(X AND 7))
```

to plot a background point. You can save a lot of time with a few enhancements to the above:

1. Substitute a variable for every constant:

```
F=255: P=256: Z=49222
```

2. Set up two arrays ahead of time:

```
FOR I=Ø TO 7: P(I)=2↑(7-I):
 M(I)=255-P(I): NEXT I
```

Then, to plot a foreground point:

```
POKE BY,PEEK(BY) OR P(X AND 7)
```

To plot a background point:

```
POKEMBY,PEEK(BY) AND M(X AND 7)
```

Of course, the ultimate in speed is obtained by skipping BASIC altogether. You can write a very simple program that reads numbers from a table and stores them in the registers XPOS, XHI, and YPOS. Then enter at label PPLOT. A little extra caution is necessary to be sure everything is set up properly.

Enhancements

The most convenient way to use BMCALC from BASIC would be something like:

```
SYS WW+88,<X expression> ,
   <Y expression> ,<C expression>
```

So to draw a vertical line, you could write:

```
FOR Y=Ø TO 199
SYS WW+88,5Ø,Y,1
NEXT Y
```

Writing such a driver is straightforward, but it takes a little more code. This technique will be covered in a future article.

Next Month

The power of machine language plotting is amplified with a machine-language line calculator program and a driver that automatically reads the values of BASIC variables X1, Y1, X2, Y2, and C. Fast line drawing makes a lot more possible, including animation of 3-D objects.

```
                ;****************************
                ;*                          *
                ;*  BIT MAP CALCULATOR WITH *
                ;*   BASIC-VARIABLE DRIVER  *
                ;*                          *
                ;*  BY LOREN W. WRIGHT      *
                ;*                          *
                ;****************************
                ;
C040            POINTR = $FB     ;ZERO-PAGE POINTER
                ;
                ;$C000-$C00F RESERVED FOR USER
                ;
                ;SYSTEM CONSTANTS
                ;
 C040           BMLOC  = $C010   ;START BIT MAP (HI)
C040            SCRHI  = BMLOC+1 ;START SCREEN MEM (HI)
 C040           MCFLAG = BMLOC+2 ;MC ON=$10--OFF=$00
 C040           PMODE  = BMLOC+3 ;PLOT MODE OR COLOR
                ;
                ;GENERAL-PURPOSE AND TEMPORARY
 C040           MASK   = BMLOC+4
 C040           TEMP   = BMLOC+5
C040            ENDHI  = TEMP
 C040           PATTRN = BMLOC+6
 C040           FCOLOR = PATTRN
                ;
                ;INFO FOR CURRENT BIT-MAP POINT
                ;
 C040           XPOS   = $C018   ;LOW 8 BITS
C040            XHI    = XPOS+1  ;9TH BIT
 C040           YPOS   = XPOS+2  ;TOP LEFT ORIGIN
 C040           YHI    = XPOS+3  ;ERROR CHECKING
 C040           COLUMN = XPOS+4  ;CHARACTER COLUMN
C040            ROW    = XPOS+5  ;CHARACTER ROW
                ;
                ;$C030-$C03F RESERVED FOR LINE CALC
                ;USAGE NEXT MONTH
                ;
                ;*
                ;* VECTORS FOR ROUTINES
                ;*
 C040                   *=   $C040
                ;
C040 4C 70 C0 INITV    JMP  INIT
                ; INITIALIZATION
 C043 4C A6 C0 PLOTV    JMP  XYPLOT
                ; USES BASIC X,Y & C TO PLOT PT
C046 4C BB C1 CALCV    JMP  BMCALC
                ; USES XPOS,XHI,YPOS
 C049 4C 82 C2 LINEV    JMP  LNDRAW
                ; USES BASIC X1,Y1,X2,Y2,C TO DRAW LINE
C04C 4C 08 C2 HCLRV    JMP  HCLEAR
                ; FILL BIT MAP WITH PATTERN
 C04F 4C 2E C2 CCLRV    JMP  CCLEAR
                ; FILL COLOR MEM WITH COLOR
C052 4C 3D C2 SET1V    JMP  SET1
                ; FILL SCREEN MSB-4 WITH COLOR
 C055 4C 4E C2 SET2V    JMP  SET2
                ; FILL SCREEN LSB-4 WITH COLOR
                ;
                ;HERE THROUGH $C06F RESERVED FOR
                ;USER'S VECTORS
                ;
                ;CALCULATES BIT MAP START FROM
                ; BANK SELECTION
                ; & BIT MAP LOCATION
                ;CALCULATES SCREEN MEM START
                ;
 C070                   *=   $C070
```

```
C070 AD 00 DD INIT     LDA  56576   ;CIA PORT A
C073 49 03             EOR  #3      ;REVERSE BITS
C075 A2 06             LDX  #6      ;0 & 1 (BANK)
C077 0A       LOOP0    ASL  A       ;SHIFT TO
C078 CA                DEX          ;POS 6 & 7
C079 D0 FC             BNE  LOOP0
C07B 8D 10 C0          STA  BMLOC
C07E 8D 11 C0          STA  SCRHI
C081 A9 08             LDA  #8
C083 2D 18 D0          AND  53272   ;BIT MAP POS
C086 F0 02             BEQ  NEXT1   ;WITHIN BANK
C088 A9 20             LDA  #%00100000  ;LOW 8K
C08A 0D 10 C0 NEXT1    ORA  BMLOC   ;SET BIT 5 IF
C08D 8D 10 C0          STA  BMLOC   ;TOP 8K
C090 AD 18 D0          LDA  53272   ;GET SCREEN LOC
C093 29 F0             AND  #%11110000
C095 4A                LSR  A       ;DIVIDE BY 4
C096 4A                LSR  A
C097 0D 11 C0          ORA  SCRHI   ;OR IN BANK #
C09A 8D 11 C0          STA  SCRHI
C09D AD 16 D0          LDA  53270
C0A0 29 10             AND  #$10    ;CHECK FOR
C0A2 8D 12 C0          STA  MCFLAG  ; MULTICOLOR
C0A5 60                RTS
                ;
                ;PLOT POINT ON BIT MAP SCREEN
                ; FROM BASIC VARIABLES X, Y, & C
                ;
C0A6 A9 58    XYPLOT   LDA  #"X"    ;SET UP BASIC
C0A8 85 45             STA  $45     ; VAR X
C0AA A9 00             LDA  #0
C0AC 85 46             STA  $46
C0AE 20 7D C1          JSR  VARSET
C0B1 A5 64             LDA  $64
C0B3 8D 19 C0          STA  XHI
C0B6 A5 65             LDA  $65
C0B8 8D 18 C0          STA  XPOS
C0BB A9 59             LDA  #"Y"    ;SET UP BASIC
C0BD 85 45             STA  $45     ; VARIABLE Y
C0BF 20 7D C1          JSR  VARSET  ;$46 IS ZERO
C0C2 A9 C7             LDA  #199    ;INVERT Y TOP
C0C4 38                SEC          ; TO BOTTOM
C0C5 E5 65             SBC  $65     ; FOR LOWER-LEFT
C0C7 8D 1A C0          STA  YPOS    ; ORIGIN
C0CA A9 00             LDA  #0
C0CC 8D 1B C0          STA  YHI
C0CF A9 43             LDA  #"C"    ;SET UP VAR C
C0D1 85 45             STA  $45     ;$46 IS STILL 0
C0D3 20 7D C1          JSR  VARSET
                ;
C0D6 AD 12 C0 PPLOT    LDA  MCFLAG  ;CHECK FOR
C0D9 D0 3D             BNE  MCPLOT  ; MULTICOLOR
                ;
C0DB 20 8B C1 HRPLOT   JSR  ERRCHK
C0DE D0 37             BNE  HRERRX
C0E0 20 BB C1          JSR  BMCALC  ;CALCULATE
                                    ; BIT MAP BYTE
C0E3 AD 18 C0          LDA  XPOS
C0E6 29 07             AND  #7
C0E8 49 07             EOR  #7      ;LEFT TO RIGHT
C0EA AA                TAX
C0EB BD 6D C1          LDA  HRTABL,X
C0EE 8D 14 C0          STA  MASK
C0F1 AC 13 C0          LDY  PMODE   ;PLOT MODE IN Y
C0F4 A2 00             LDX  #0
C0F6 A1 FB             LDA  (POINTR,X)  ;GET CURRENT
                                    ; BIT-MAP BYTE
C0F8 C0 02    TOGGLE   CPY  #2      ;TOGGLE MODE
C0FA D0 05             BNE  FGPLOT
C0FC 4D 14 C0          EOR  MASK
C0FF 90 14             BCC  FINIS
```

```
C1Ø1 CØ Ø1    FGPLOT   CPY  #1       ;FOREGROUND MODE
C1Ø3 DØ Ø5             BNE  BGPLOT
C1Ø5 ØD 14 CØ          ORA  MASK
C1Ø8 9Ø ØB             BCC  FINIS
                ;
C1ØA CØ ØØ    BGPLOT   CPY  #Ø       ;BACKGND/ERASE MODE
C1ØC DØ Ø7             BNE  FINIS    ;NO CHANGE IF > 2
C1ØE AD 14 CØ          LDA  MASK     ; OR < Ø
C111 49 FF             EOR  #%11111111
C113 21 FB             AND  (POINTR,X)
                ;
C115 81 FB    FINIS    STA  (POINTR,X)  ;STORE NEW
C117 6Ø       HRERRX   RTS              ; VERSION
                ;
C118 2Ø AD C1 MCPLOT   JSR  MCERCK
C11B DØ 3E             BNE  MCERRX
C11D ØE 18 CØ          ASL  XPOS     ;SEND BMCALC 2*X
C12Ø 2E 19 CØ          ROL  XHI
C123 2Ø BB C1          JSR  BMCALC
C126 4E 19 CØ          LSR  XHI      ;AND RESTORE
C129 6E 18 CØ          ROR  XPOS
C12C AD 18 CØ          LDA  XPOS
C12F 29 Ø3             AND  #3
C131 49 Ø3             EOR  #3       ;MASK CONTAINS
C133 8D 14 CØ          STA  MASK     ; BIT-PAIR POS.
C136 AD 13 CØ          LDA  PMODE    ;Ø-3 FOR COLORS,
C139 C9 Ø4             CMP  #4       ; 4 TO TOGGLE
C13B FØ 1F             BEQ  MCTOGL
C13D BØ 1C             BCS  MCERRX
C13F AE 14 CØ          LDX  MASK
C142 FØ Ø5             BEQ  MCNEXT
C144 ØA       MCLOOP   ASL  A        ;SHIFT PMODE TO
C145 ØA                ASL  A        ; PROPER BIT PAIR
C146 CA                DEX
C147 DØ FB             BNE  MCLOOP
                ;
C149 8D 15 CØ MCNEXT   STA  TEMP     ;NEW PIXEL DATA
C14C A1 FB             LDA  (POINTR,X)  ;GET CURRENT
C14E AE 14 CØ          LDX  MASK         ; BM BYTE
C151 3D 75 C1          AND  SVTABL,X  ;SAVE OTHER
C154 ØD 15 CØ          ORA  TEMP      ; 3 BIT PAIRS
C157 A2 ØØ             LDX  #Ø
C159 81 FB             STA  (POINTR,X)
                ;
C15B 6Ø       MCERRX   RTS
                ;
C15C A2 ØØ    MCTOGL   LDX  #Ø
C15E A1 FB             LDA  (POINTR,X)
C16Ø AE 14 CØ          LDX  MASK
C163 3D 79 C1          AND  STTABL,X
C166 5D 79 C1          EOR  STTABL,X
C169 A2 ØØ             LDX  #Ø
C16B FØ DC             BEQ  MCNEXT   ;ALWAYS
                ;
C16D Ø1 Ø2 Ø4 HRTABL   .BYTE $Ø1,$Ø2,$Ø4,$Ø8,$1Ø
C172 2Ø 4Ø 8Ø          .BYTE $2Ø,$4Ø,$8Ø
C175 FC F3 CF SVTABL   .BYTE $FC,$F3,$CF,$3F
C179 Ø3 ØC 3Ø STTABL   .BYTE $Ø3,$ØC,$3Ø,$CØ
                ;
                ;VARIABLE NAME IN $45,$46
                ;RETURNS INTEGER IN $64(HI),$65(LO)
                ;
C17D 2Ø E7 BØ VARSET   JSR  $BØE7    ;FIND VARIABLE
C18Ø A5 47             LDA  $47      ; (ROM ROUTINE)
C182 A4 48             LDY  $48      ;LOAD FAC #1 W/
C184 2Ø A2 BB          JSR  $BBA2    ; VALUE (ROM)
C187 2Ø 9B BC          JSR  $BC9B    ;FAC-TO-INT
C18A 6Ø                RTS      ; H-$64,L-$65 (ROM)
                ;
C18B AD 19 CØ ERRCHK   LDA  XHI
C18E C9 Ø1             CMP  #1
C19Ø 9Ø Ø9             BCC  YCHEK
C192 DØ 16             BNE  ERRTRN
C194 AD 18 CØ          LDA  XPOS
C197 C9 4Ø             CMP  #319-256+1
C199 BØ ØF             BCS  ERRTRN
                ;
C19B AD 1B CØ YCHEK    LDA  YHI
C19E DØ ØA             BNE  ERRTRN
C1AØ AD 1A CØ          LDA  YPOS
C1A3 C9 C8             CMP  #199+1
C1A5 BØ Ø3             BCS  ERRTRN
                ;
C1A7 A9 ØØ    NOERR    LDA  #Ø
C1A9 6Ø                RTS
                ;
C1AA A9 FF    ERRTRN   LDA  #$FF
C1AC 6Ø                RTS
                ;
C1AD AD 19 CØ MCERCK   LDA  XHI
C1BØ DØ F8             BNE  ERRTRN
C1B2 AD 18 CØ          LDA  XPOS
C1B5 C9 AØ             CMP  #159+1
C1B7 BØ F1             BCS  ERRTRN
C1B9 9Ø EØ             BCC  YCHEK
                ;
                ;CALCULATE BYTE FROM XHI,XPOS & YPOS
                ;COLUMN & ROW MAINTAINED
                ; FOR CHAR & COLOR CALCULATIONS
                ;
C1BB AD 18 CØ BMCALC   LDA  XPOS
C1BE 8D 1C CØ          STA  COLUMN
C1C1 AD 1A CØ          LDA  YPOS
C1C4 8D 1D CØ          STA  ROW
C1C7 AD 19 CØ          LDA  XHI
C1CA A2 Ø3             LDX  #3       ;INTEGER DIVIDE
C1CC 4A       LOOPB    LSR  A        ; BY 8
C1CD 6E 1C CØ          ROR  COLUMN   ;9TH BIT OF
C1DØ 4E 1D CØ          LSR  ROW      ; XPOS FROM XHI
C1D3 CA                DEX
C1D4 DØ F6             BNE  LOOPB
C1D6 A9 ØØ             LDA  #Ø
C1D8 85 FC             STA  POINTR+1
C1DA A2 Ø6             LDX  #6
C1DC AD 1D CØ          LDA  ROW      ;MULTIPLY BY 64
C1DF ØA       LOOPC    ASL  A
C1EØ 26 FC             ROL  POINTR+1
C1E2 CA                DEX
C1E3 DØ FA             BNE  LOOPC
C1E5 85 FB             STA  POINTR
C1E7 AD 18 CØ          LDA  XPOS
C1EA 29 F8             AND  #%11111ØØØ  ;SAME AS
C1EC 18                CLC              ; 8 * COLUMN
C1ED 65 FB             ADC  POINTR
C1EF 85 FB             STA  POINTR
C1F1 A5 FC             LDA  POINTR+1
C1F3 6D 1D CØ          ADC  ROW      ;SAME AS +256*ROW
C1F6 6D 19 CØ          ADC  XHI
C1F9 ØD 1Ø CØ          ORA  BMLOC    ;BITS 5, 6, & 7
C1FC 85 FC             STA  POINTR+1 ; FROM INIT
C1FE AD 1A CØ          LDA  YPOS
C2Ø1 29 Ø7             AND  #%ØØØØØ111  ;GET LINE
C2Ø3 Ø5 FB             ORA  POINTR   ; WITHIN CHAR ROW
C2Ø5 85 FB             STA  POINTR
C2Ø7 6Ø                RTS
                ;
                ;*CLEARING ROUTINES
                ;*
                ;*INIT MUST HAVE BEEN PERFORMED
                ;* AND $CØ18 MUST CONTAIN COLOR
                ;* OR PATTERN
                ;
```

```
C2Ø8 AD 1Ø CØ HCLEAR   LDA  BMLOC   ;ENTRY TO SET BIT
C2ØB 85 FC             STA  POINTR+1  ;MAP TO PATTERN
C2ØD 18                CLC           ;SET UP POINTR
C2ØE 69 2Ø             ADC  #$2Ø      ; & ENDHI
C21Ø 8D 15 CØ          STA  ENDHI
C213 A9 ØØ             LDA  #Ø
C215 85 FB             STA  POINTR
                ;
C217 AØ ØØ     HMAIN   LDY  #Ø
C219 AD 16 CØ HLOOP    LDA  PATTRN  ;PAGE CLEAR USED
C21C 91 FB     INLOOP  STA  (POINTR),Y  ; BY HCLEAR
C21E C8                INY           ; AND CCLEAR
C21F DØ FB             BNE  INLOOP
C221 A5 FC             LDA  POINTR+1  ;ADVANCE PAGE
C223 18                CLC
C224 69 Ø1             ADC  #1
C226 85 FC             STA  POINTR+1
C228 CD 15 CØ          CMP  ENDHI
C22B DØ EC             BNE  HLOOP
C22D 6Ø                RTS
                ;
C22E A9 D8     CCLEAR  LDA  #$D8    ;ENTRY TO SET
C23Ø 85 FC             STA  POINTR+1  ; COLOR MEM TO
C232 A9 ØØ             LDA  #Ø       ; COLOR-MC COLOR 11
C234 85 FB             STA  POINTR  ;SETUP POINTER
C236 A9 DC             LDA  #$DC     ; & ENDHI
C238 8D 15 CØ          STA  ENDHI
C23B DØ DA             BNE  HMAIN   ;ALWAYS
                ;
C23D AD 16 CØ SET1     LDA  FCOLOR  ;ENTRY TO SET MC
C24Ø ØA                ASL  A        ; COLOR Ø1 &
C241 ØA                ASL  A        ; HR BACKGROUND
C242 ØA                ASL  A
C243 ØA                ASL  A
C244 8D 16 CØ          STA  FCOLOR
C247 A9 ØF             LDA  #%ØØØØ1111  ;TO PRESERVE
C249 8D 14 CØ          STA  MASK       ; LOW 4 BITS
C24C DØ Ø5             BNE  SETCOLOR  ;ALWAYS
                ;
                ; ENTRY TO SET MC COLOR 1Ø & HR
C24E A9 FØ     SET2    LDA  #%1111ØØØØ
C25Ø 8D 14 CØ          STA  MASK
                ;
C253 AD 11 CØ SETCOLOR LDA  SCRHI     ;SET UP POINTR
C256 85 FC             STA  POINTR+1  ; AND ENDHI
C258 18                CLC
C259 69 Ø3             ADC  #3
C25B 8D 15 CØ          STA  ENDHI
C25E A9 ØØ             LDA  #Ø
C26Ø 85 FB             STA  POINTR
C262 A8                TAY
                ;
C263 B1 FB     CLOOP   LDA  (POINTR),Y  ;TAKE WHAT'S
C265 2D 14 CØ          AND  MASK     ; THERE, KEEP
C268 ØD 16 CØ          ORA  FCOLOR  ;THE OTHER
C26B 91 FB             STA  (POINTR),Y  ; COLOR, CHANGE
C26D C8                INY           ; PROPER 4 BITS
C26E FØ ØD             BEQ  NEXTPG   ; AND PUT IT BACK
C27Ø CØ E8             CPY  #$E8     ;SCREEN+$3E7
C272 DØ EF             BNE  CLOOP    ; IS END
C274 A5 FC             LDA  POINTR+1  ;TEST FOR
C276 CD 15 CØ          CMP  ENDHI     ; LAST PAGE
C279 DØ E8             BNE  CLOOP    ;CONTINUE IF NOT
C27B FØ Ø4             BEQ  DONE
                ;
C27D E6 FC     NEXTPG  INC  POINTR+1  ;ADVANCE PAGE
C27F DØ E2             BNE  CLOOP
                ;
C281 6Ø        DONE    RTS
                ;
C282 6Ø        LNDRAW  RTS           ;NEXT MONTH!
```

Sine Wave Demo

```
1Ø WW=49152
2Ø GOSUB 8ØØØ: REM SET UP SCREEN
3Ø SYS WW+64: REM INIT
4Ø POKE WW+22,Ø: SYS WW+76: REM CLEAR BIT-MAP
5Ø POKE WW+22,Ø: SYS WW+82: REM FGD=Ø
6Ø POKE WW+22,1: SYS WW+85: REM BGD=1
1ØØ C=1: P=3.14159265
11Ø FOR X=Ø TO 319
12Ø Y=1ØØ+INT(SIN(X*P/16Ø)*95+.5)
13Ø SYS WW+67: REM PLOT POINT
14Ø NEXT X
9ØØ GET T$:IF T$="" THEN 9ØØ
91Ø GOSUB 81ØØ
999 STOP
8ØØØ REM HI-RES SETUP
8Ø1Ø POKE 56578,PEEK(56578) OR 3
8Ø2Ø POKE 56576,PEEK(56576) AND 252 OR 2
8Ø3Ø POKE 53272,PEEK(53272) AND 7 OR 12Ø
8Ø4Ø POKE 53265,PEEK(53265) OR 32
8Ø5Ø RETURN
81ØØ REM RESTORE CHAR SCREEN
811Ø POKE 56578,PEEK(56578) OR 3
812Ø POKE 56576,PEEK(56576) OR 3
813Ø POKE 53272,PEEK(53272) AND 7 OR 16
814Ø POKE 53265,PEEK(53265) AND 223
815Ø RETURN
```

Machine Language Loops

by Chris Williams
Ogden, Utah

Misconceptions About Machine Language Loops — Exposed

This is the latest in an informal series of articles on speed in assembly language programming. The first article was a presentation of techniques. This one is a discussion of a popular misconception about machine language loops.

Readers who are experienced machine language programmers are probably frowning right now in reaction to the title and that's good. It was meant to get their attention. It is they who are wrong about loops.

How? Well, like any other technical type, machine language programmers tend to use educated intuition in deciding how to perform a given task. That intuition is the product of experience. The solutions they've encountered in the past tend to get selected as the best solution for the problem at hand.

Fine. And yes, I know that's obvious. But it has a subtle ramification that quite literally has swept the industry. You see, there's a problem. Suppose those solutions encountered in the past were wrong?

Examine, if you will, the following sequence of 6502 assembly language instructions.

```
LDA  $VAL
STA  $VALSTR
LDA  $VAL+1
STA  $VALSTR+1
LDA  $VAL+2
STA  $VALSTR+2
LDA  $VAL+3
STA  $VALSTR+3
```

You've probably never seen that sort of thing before, especially if it had gone out to $VAL+9 or $VAL+10 or farther.

No, most likely you've seen that operation done like so:

```
      LDX #Ø4
LOOP  LDA $VAL,X
      STA $VALSTR,X
      DEX
      BNE LOOP
...
```

And some people would use ...

```
      LDX #ØØ
LOOP  LDA $VAL,X
      STA $VALSTR,X
      INX
      CPX #Ø4
      BNE LOOP
```

... though they shouldn't and, if they'd read my previous article on speed and counting down, they would have known better.

But back to the point. Which of the two ways of LOADing and STOREing do you prefer? Sure, you like the second one. Loops are elegant. They're popular. They're easier to type in. And your intuition tells you they're just plain superior.

Let's see. First, let's examine the memory storage in Figure 1.

The results there show a healthy memory savings for the loop. For a speed comparison, now look at Figure 2.

Hmmm, the byte ratio is 24/11 (24/13 for the wrong way) in favor of the loop and the speed ratio is 62/32 in favor of sequence code. Let's observe here that, if there is no inordinate value placed on either speed or memory (i.e., they're equally valuable), the larger memory ratio advantage of the loop is probably compelling and that would be the way to go. All who preferred the loop may congratulate themselves. Their intuition in this case was correct.

Now then, suppose there was only one LOAD-STORE pair as shown in Figure 3. Would anyone use a loop here?

```
                    Bytes
                    Used

      LDA $VAL        3
      STA $VALSTR     3
      LDA $VAL+1      3
      STA $VALSTR+1   3
      LDA $VAL+2      3
      STA $VALSTR+2   3
      LDA $VAL+3      3
      STA $VALSTR+3   3
                     ----
                     24 BYTES


And for the loop:


      LDX #Ø4         2
LOOP  LDA $VAL,X      3
      STA $VALSTR,X   3
      DEX             1
      BNE LOOP        2
      ...
                     ---
                     11 BYTES


Or, if you insist on doing it wrong:


      LDX #ØØ         2
LOOP  LDA $VAL,X      3
      STA $VALSTR,X   3
      INX             1
      CPX #Ø4         2
      BNE LOOP        2
      ...
                     ---
                     13 BYTES


Figure 1 Memory Usage
```

```
Machine Cycles


     LDA $VAL        4
     STA $VALSTR     4
     LDA $VAL+1      4
     STA $VALSTR+1   4
     LDA $VAL+2      4
     STA $VALSTR+2   4
     LDA $VAL+3      4
     STA $VALSTR+3   4
                    ---
                    32 CYCLES


And for the loop:


     LDX #Ø4         2
LOOP LDA $VAL,X      4
     STA $VALSTR,X   5
     DEX             2
     BNE LOOP        4
     ...
                    ---
                2+(15*4)=62 CYCLES
```

Figure 2 Speed in Machine Cycles

I hope not. The numbers clearly show the loop to be inferior in both memory used and speed.

So, the whole issue comes down to this question. At what point should you stop writing LOAD-STORE pairs and start writing loops?

I've asked this question of several assembly language competent friends and their answers are interesting, mainly because they were all the same.

"Use a loop when there's three or more pairs," they said, almost as one. "Three is the magic number."

I'll bet you agree.

Sorry. The right answer is two. Examine Figure 4.

You pass memory breakeven at 2 pairs and, indeed, have a one byte advantage there. The speed ratio is 2:1. At 3 pairs the memory advantage continues to grow and the speed ratio shrinks almost not at all. Since the speed ratio is essentially the same, the memory factor is decisive, and since we passed equality there at 2 pairs, then clearly that was the point at which to switch over to loops, not at 3 pairs.

This failure in intuition is disconcerting. One grows to depend on it in technical fields. It's usually not that deceptive. I suspect it fails us in this case because a loop requires us to type 5 lines of code for 2 pairs as opposed to 4 lines of code for a sequence, even though the memory requirement is less. It's laziness changing our minds and warping our judgement here, not logic. Be aware of it.

```
                  Bytes    Machine
                  Used      Cycles
     LDA $VAL      3          4
     STA $VALSTR   3          4
     ...
                  ---        ---
                  6 BYTES    8 CYCLES


The loop would be:


     LDX #1        2          2
LOOP LDA $VAL      3          4
     STA $VALSTR   3          4
     DEX           1          2
     BNE LOOP      2          4
     ...
                  ---        ---
                  11 BYTES   16 CYCLES
```

Figure 3 The Single LOAD-STORE Pair

For three LOAD-STORE pairs:

```
                  Bytes    Machine
                  Used      Cycles
     LDA $VAL      3          4
     STA $VALSTR   3          4
     LDA $VAL+1    3          4
     STA $VALSTR+1 3          4
     LDA $VAL+2    3          4
     STA $VALSTR+2 3          4
     ...
                  ---        ---
                  18 BYTES   24 CYCLES


The loop would be:


     LDX #Ø3       2          2
LOOP LDA $VAL,X    3          4
     STA $VALSTR,X 3          5
     DEX           1          2
     BNE LOOP      2          4
     ...
                  ---        ---
                  11 BYTES   2+(15*3)=47 CYCLES
```

For two LOAD-STORE pairs:

```
     LDA $VAL      3          4
     STA $VALSTR   3          4
     LDA $VAL+1    3          4
     STA $VALSTR+  3          4
     ...
                  ---        ---
                  12 BYTES   16 CYCLES


The loop would be:


     LDX #Ø2       2          2
LOOP LDA $VAL,X    3          4
     STA $VALSTR,X 3          5
     DEX           1          2
     BNE LOOP      2          4
     ...
                  ---        ---
                  11 BYTES   2+(15*2)=32 CYCLES
```

Figure 4 Two vs Three Pair

Interactive Input Utility

by Mike Dougherty
Littleton, Colorado

You Can Improve the Usefulness of Your FORTH Programs by Adding this Interactive Input Utility

Introduction

FORTH contains a rich vocabulary to output data and information. Words such as .R, #, TYPE and ." allow great flexibility for printing data. Unfortunately, user input is handled primarily through the text interpreter. Turnkey applications and users not familiar with FORTH often require a more interactive approach. The Input Utility, Listing 1, defines MENU and PROMPT to supply some of this missing interaction.

Menu

The utility MENU, Screens 55-67, allows a FORTH application to display lines 1-15 of a disk Screen as a menu of choices. A choice is selected from the menu via the keyboard consol switches. Line 0 of the menu Screen is reserved for MENU parameters. The entire menu Screen may be created or modified with a FORTH text editor.

A simple menu, Screen # 26, is shown in Figure 1. To use this Screen as a menu, execute:

```
26 MENU
```

Upon execution, the current menu selection will be set to the first menu selection. Each non-blank character in the current menu selection is highlighted by inverse video. The Atari operating system shadow register, CHACTL, located at memory address 755, is used to blink the highlighted characters by the word BLINK. The user changes the current menu selection by pressing the SELECT or OPTION consol switch. When the START consol switch is pressed, MENU returns the value of the current menu selection, 1 to n, where n is the total number of menu selections.

MENU uses two single precision numbers located in an ASCII free text format in line 0 of the menu Screen. The first number, TOP-MENU, defines the menu Screen line number (1 through 15) of the first menu selection. The second number, BOT-MENU, defines the menu Screen line number

```
SCR # 26
  0   9   12
  1       **********************
  2       *                    *
  3       * GRADES SELECTION   *
  4       *                    *
  5       **********************
  6
  7   SELECT one of the following:
  8
  9      1) Read Class from Disk
 10      2) Class Modification
 11      3) Report Generation
 12      4) Write Class to Disk
 13
 14
 15 Press START to choose SELECTion
```

Figure 1 Simple Menu

(TOP-MENU through 15) of the last selection. The total number of menu selections is BOT-MENU - TOP-MENU + 1 . These selection limits are read from the menu Screen by the word SET-LIMITS. MENU makes the assumption that each menu selection will take only one video line to display. When I use MENU, I set the left margin offset, LEFT-OFFSET, to 2 and limit each menu selection to a 32 character line. (My FORTH Screen Editor manipulates half lines of 32 characters with particular ease.) For vertical spacing, the top margin offset, TOP-OFFSET, is set to 2. Extensions to MENU could allow the spacing offsets to be read from the first menu Screen line along with the selection parameters.

MENU relieves the FORTH application of the details of displaying text menus. Instead, the application is only concerned with responding to the user selection. Further, the menu wording may be modified without re-LOADing the application.

Prompt

The utility PROMPT, Screens 68-72, allows a FORTH application to prompt a user to input a single precision integer within a specified range. The prompt is repeated until a value within that range is entered. APX fig-FORTH 1.1 already defines the word PROMPT to print the FORTH "ok" message. Since I never need this function, I let my utility redefine PROMPT. For any user requiring the old definition of PROMPT, the Input Utility version should be renamed.

The prompt text is created with the '' defining word, Screens 68 and 69. For example, to prompt for examination grades, a prompt named TEST may be defined:

```
1 100 '' TEST Exam Grade (1-100): ''
```

The '' defining word will use the next word in the input stream, TEST, to create a dictionary entry by the BUILDS portion of the '' definition. The rest of the BUILDS compiles the input limits from the stack and a dimensioned string of characters into the dictionary. The results of the above PROMPT string is illustrated in Figure 2. Subsequent execution of TEST leaves the prompt text address, prompt length, minimum and maximum on the stack as defined by the DOES portion of the '' defining word.

The following uses TEST as a prompt and returns a value in the range of 1 to 100 inclusively.

```
TEST PROMPT
```

The returned value will be a single precision number on top of the data stack. PROMPT will not return until a user number is entered within the specified range of 1 to 100. In general, to define and use the prompt XYZ to input a number between n1 and n2:

```
n1 n2 '' XYZ ... prompt text ...''
XYZ PROMPT
```

The use of PROMPT relieves the FORTH application of the burden of performing range validation for each input.

Conclusion

The utilities MENU and PROMPT were defined to solve specific problems I had with my applications. Obviously, they can be modified to fit each user's own needs. The main idea remains to modify past FORTH definitions to fit into new situations, saving programming work and time.

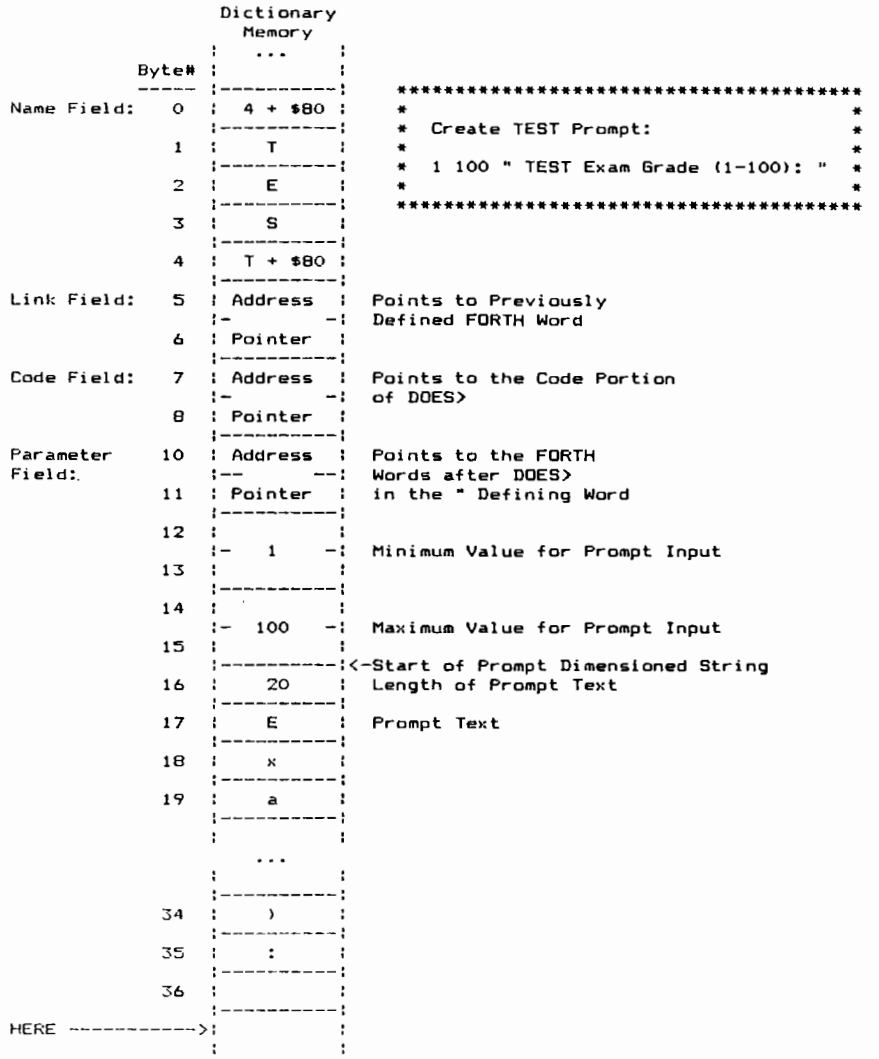


Figure 2 Dictionary Memory of TEST Prompt

```
SCR # 54
 Ø ( INPUT UTILITY                    1133 BYTES )
 1
 2 ( To Load the utility:             )
 3 (  54 LOAD                         )
 4 (                                  )
 5 ( Usage:                           )
 6 (                                  )
 7 ( To execute Screen #26 menu:      )
 8 (  26 MENU                         )
 9 (                                  )
10 ( To define and use XYZ prompt:    )
11 (  min max                         )
12 (  " XYZ ...prompt text..."        )
13 (  XYZ PROMPT                      )
14
15 —>
```

```
SCR # 55
 Ø ( ADDR OF LINE OF MENU - FORCES  A DISK READ IF NEEDED )
 1
 2 26 VARIABLE MENU#          ( Current menu number )
 3
 4 : ADDR-MENU                ( line# —— addr )
 5     MENU# @ (LINE)         ( Read and get address of line )
 6     DROP ;                 ( Drop the length of 64 )
 7
 8 —>
```

```
SCR # 56
 Ø ( SCAN ASCII STRING FOR NEXT NUMBER )
 1
 2 : SCAN-NUMBER              ( addr —— addr n )
 3     HERE 34 BLANKS         ( Clear dest for dim string )
 4     BL ENCLOSE             ( Find numerical string limits )
 5     >R                      ( Save the next string offset )
 6     OVER - DUP HERE C!     ( Save length in dim string )
 7     ROT >R R                ( Save a copy of the orig addr )
 8     ROT +                  ( Adr of the start of string )
 9     HERE 1+ ROT            ( Addr to move the string )
10     CMOVE                  ( Move dim string to HERE )
11     R> R> +                  ( Address of next string )
12     HERE NUMBER DROP ;     ( Convert to single prec num )
13
14 —>
```

```
SCR # 57
  Ø ( INPUT SELECTION LIMITS FROM MENU'S FIRST LINE )
  1
  2 Ø VARIABLE TOP-MENU        ( Screen line# of 1st  choice )
  3 Ø VARIABLE BOT-MENU        ( Screen line# of last choice )
  4 Ø VARIABLE LINE#           ( Screen line# of current )
  5
  6 : SET-LIMITS               ( —— )
  7     Ø ADDR-MENU            ( Read menu line Ø, get addr )
  8     SCAN-NUMBER TOP-MENU !( The top line of the options  )
  9     TOP-MENU @ LINE# !     ( Sae for current line # )
 1Ø     SCAN-NUMBER BOT-MENU !( The bottom line of the opts )
 11     DROP ;                 ( Drop the final addr )
 12
 13 —>


SCR # 58
  Ø ( SCREEN DISPLAY FUNCTIONS )
  1
  2 : CLEAR-SCREEN             ( —— )
  3     125 EMIT ;             ( Output SHFT CLEAR key code )
  4
  5 : CRS                      ( n —— )
  6     -DUP IF                ( If more than Ø CR s )
  7       Ø DO                 ( Output n carriage returns )
  8         CR
  9       LOOP
 1Ø     ENDIF ;
 11
 12 —>


SCR # 59
  Ø ( DISPLAY MENU )
  1
  2 2 VARIABLE TOP-OFFSET      ( Lines from top of video )
  3 2 VARIABLE LEFT-OFFSET     ( Spaces from the left of video)
  4
  5 : DISPLAY-MENU             ( n —— )
  6     MENU# !                ( Save the current menu number )
  7     SET-LIMITS             ( Set the menu limit params )
  8     CLEAR-SCREEN           ( Erase the video screen )
  9     TOP-OFFSET @ CRS       ( Lines from the display top )
 1Ø     16 1 DO                ( For each line in the menu )
 11       I ADDR-MENU          ( Get the address of the line )
 12       64 -TRAILING         ( Trim the trailing blanks )
 13       LEFT-OFFSET @ SPACES TYPE ( Output for user )
 14       CR
 15     LOOP ;     —>
```

```
SCR # 6Ø
  Ø ( INVERSE CHARACTER BLINK )
  1
  2 755 CONSTANT CHACTL        ( OS shadow reg for display )
  3  2Ø CONSTANT RTC           ( OS real Time Clock, lsb )
  4   8 VARIABLE SOLID-BIT
  5   4 VARIABLE BLINK-BIT
  6
  7 : BLINK-INVERSE            ( —— )
  8     3 CHACTL C!            ( Turn off the inverse chars )
  9     BEGIN                  ( Wait until BLINK-BIT a one )
 1Ø       RTC @ BLINK-BIT @ AND
 11     UNTIL
 12     2 CHACTL C! ;          ( Turn on the inverse chars )
 13
 14 —>


SCR # 61
  Ø ( TIMING FOR INVERSE BLINK, READ THE CONSOL SWITCH )
  1
  2 : BLINK                        ( — )
  3     RTC @ SOLID-BIT @ AND IF   ( If SOLID-BIT is a one )
  4       BLINK-INVERSE            ( Then blink the inverse chars )
  5       BEGIN                    ( Wait until SOLID-BIT a zero )
  6         RTC @ SOLID-BIT @ AND
  7       Ø= UNTIL
  8     ENDIF ;
  9
 1Ø : GET-CONSOL                   ( —— consol )
 11     BLINK                      ( Blink inverse if needed )
 12     53279 C@                   ( Get the consol H/W register )
 13     7 XOR ;                    ( Convert to positive logic )
 14
 15 —>


SCR # 62
  Ø ( SAMPLE THE CONSOL KEYS )
  1
  2 : READ-CONSOL              ( —— switch# )
  3     BEGIN                  ( Wait until switch released )
  4       GET-CONSOL Ø=
  5     UNTIL
  6     BEGIN                  ( Wait until switch pressed )
  7       GET-CONSOL
  8     UNTIL
  9     GET-CONSOL ;           ( Return the switch value now )
 1Ø
 11 —>
```

```
SCR # 63
  Ø ( GENERATE THE DISPLAY SCREEN ADDRESS )
  1
  2 : SCREEN                  ( -- addr )
  3     1Ø6 C@                ( Get the hi mem page number )
  4     256 *                 ( Convert to address )
  5     96Ø - ;               ( Beginning of graphics Ø )
  6
  7 -->
```

```
SCR # 64
  Ø ( SCREEN FUNCTIONS )
  1
  2 : INVERT-VIDEO            ( -- )
  3     SCREEN                ( Get starting addr of video )
  4     LINE# @ 1 -           ( Get the relative line # )
  5     TOP-OFFSET @ +        ( Adust for offset )
  6     4Ø * +                ( 4Ø bytes per line )
  7     4Ø OVER + SWAP DO     ( Do between line addr limits )
  8       I C@                ( Get a byte from the screen )
  9       -DUP IF             ( If not a video screen blank )
 1Ø        128 XOR           ( Flip the video bit )
 11        I C!              ( Restore the inverted char )
 12      ENDIF
 13    LOOP ;    -->
```

```
SCR # 65
  Ø ( UPDATE A NEW USER SELECTION )
  1
  2 : CHANGE-SELECTION        ( -- )
  3     INVERT-VIDEO          ( Invert current back to norm )
  4     LINE# @ 1+            ( Update the line to highlight )
  5     DUP BOT-MENU @ > IF   ( If beyond the menu bottom )
  6       DROP                ( Drop out of range value )
  7       TOP-MENU @          ( Back to the top of the menu )
  8     ENDIF
  9     LINE# !               ( Set LINE# to new value )
 1Ø     INVERT-VIDEO ;        ( Invert the new current line )
 11
 12 -->
```


```
SCR # 65
  Ø ( UPDATE A NEW USER SELECTION )
  1
  2 : CHANGE-SELECTION        ( -- )
  3     INVERT-VIDEO          ( Invert current back to norm )
  4     LINE# @ 1+            ( Update the line to highlight )
  5     DUP BOT-MENU @ > IF   ( If beyond the menu bottom )
  6       DROP                ( Drop out of range value )
  7       TOP-MENU @          ( Back to the top of the menu )
  8     ENDIF
  9     LINE# !               ( Set LINE# to new value )
 1Ø     INVERT-VIDEO ;        ( Invert the new current line )
 11
 12 -->
```


```
SCR # 66
  Ø ( MENU SELECTION FUNCTION )
  1
  2 : GET-SELECTION           ( -- n )
  3     INVERT-VIDEO          ( Highlight the first option)
  4     BEGIN                 ( Wait until START consol )
  5       READ-CONSOL DUP IF  ( If a consol switch pressed )
  6         1 = IF            ( If the switch is START )
  7           1               ( Flag exit from the loop )
  8         ELSE              ( The SELECT/OPTION sw pressed )
  9           CHANGE-SELECTION( Change the user selection )
 1Ø           Ø               ( Flag the loop to continue )
 11         ENDIF
 12       ENDIF
 13     UNTIL                 ( Loop until START pressed )
 14     LINE# @ TOP-MENU @ - 1;( Return 1-n based on inverse )
 15 -->
```

```
SCR # 67
  Ø ( GENERAL MENU DRIVER )
  1
  2 : MENU                    ( menu# -- option# )
  3     752 C@ SWAP           ( Save the current cursor mode )
  4     1 752 C!              ( Inhibit the screen cursor )
  5     DISPLAY-MENU          ( Display the menu )
  6     GET-SELECTION         ( Get the user's selection )
  7     SWAP 752 C! ;         ( Restore the cursor mode )
  8
  9 -->
```

```
SCR # 68
 Ø ( CREATE PROMPT DEFINING WORD: min max " name string...text" )
 1
 2 : "                          ( Define a prompt )
 3     <BUILDS                   ( Compile name into dictionary )
 4       SWAP , ,                ( Compile min,max into dict )
 5       TIB @ IN @ +            ( Addr of string text )
 6       ASCII " ENCLOSE         ( Find string numerical limits )
 7       >R                      ( Save the next string offset )
 8       OVER - DUP C,           ( Copile length into dict )
 9       >R                      ( Save length of prompt )
10       +                       ( Start of non-blank string )
11       R> Ø DO                 ( For each character in string )
12         DUP C@ C,             ( Compile into dictionary )
13         1+                    ( Next char address )
14       LOOP
15 -->

SCR # 69
 Ø ( CREATE STRING DEFINING WORD, CONT'D )
 1
 2       DROP
 3       R> IN +!                ( Set system TIB offset )
 4
 5     DOES>                     ( -- addr length )
 6       DUP @  >R 2 +           ( Get the prompt minimum, save )
 7       DUP @  >R 2 +           ( Get the prompt maximum, save )
 8       DUP C@                  ( Get the dim string length )
 9       SWAP 1+ SWAP            ( Point to prompt text )
10       R> R> SWAP ;             ( Stack limits in min,max order)
11
12 -->

SCR # 70
 Ø ( TEST RANGE OF INPUT VALUE )
 1
 2 Ø VARIABLE PROMPT-MIN      ( Minimum value allowed )
 3 Ø VARIABLE PROMPT-MAX      ( Maximum value allowed )
 4
 5 : ?RANGE                           ( n -- 1 )
 6     DUP PROMPT-MAX @ 1+ < IF        ( Ifn is maximum or below )
 7       DUP PROMPT-MIN @ 1 - > IF ( If n is minimum or above )
 8         1                          ( Set flag to true )
 9       ELSE                         ( n is below minimum )
10         DROP Ø                     ( Set flag to false  )
11       ENDIF
12     ELSE                           ( n is above maximum )
13       DROP Ø                       ( Set flag to false  )
14     ENDIF ;
15 -->
```

```
SCR # 71
 Ø ( INPUT A SET OF NUMBERS FROM A SINGLE LINE OF INPUT )
 1
 2 : INPUT-NUMBER             ( m --- n1 n2 ... nm )
 3     PAD 8Ø EXPECT          ( Input the user text line )
 4     PAD                    ( Address of ASCII string )
 5     SWAP Ø DO              ( For each of the m numbers )
 6       DUP C@ IF            ( If not the end of line, null)
 7         SCAN-NUMBER        ( Convert next numerical string)
 8       ELSE                 ( ASCII text used up )
 9         Ø                  ( Deault to a zero value )
10       ENDIF
11       SWAP
12     LOOP
13     DROP ;
14
15 -->

SCR # 72
 Ø ( PROMPT USER FOR INPUT )
 1
 2 : PROMPT                   ( addr length min max -- n )
 3     PROMPT-MAX !           ( Save limits for ?RANGE )
 4     PROMPT-MIN !
 5     BEGIN                  ( Repeat until input in range )
 6       CLEAR-SCREEN         ( Start with a clean screen )
 7       2DUP TYPE            ( Show the prompt )
 8       1 INPUT-NUMBER       ( Get a single number )
 9     ?RANGE UNTIL           ( Lop until in range )
10     SWAP DROP              ( Drop copies of addr,length )
11     SWAP DROP
12     CLEAR-SCREEN ;         ( Clan up results )
13
14 ;S
15
```

Mike Dougherty has written a number of FORTH utilities that have/will appear in MICRO, including *Structure Trees in FORTH, FORTH LinearTable Interpolation, FORTH Disk Source Utility*. His 'guest editorial' was featured in last month's issue. He has worked in the software field since 1977 and is currently a Software Engineer with Martin Marietta Denver Aerospace in Colorado.

Database Management Systems

by Sanjiva K. Nath
San Francisco, California

Detailed Discussion of the Major Components that make up a Database Management Package

Introduction

Computers are able to process data at incredible speeds. They can also store vast amounts of information in storage media a miniscule fraction of the size of filing cabinets. We refer, of course, to disks, cassettes and diskettes. Computers can organize information, sort it and present it to you in any order that you specify, all at a fraction of the time that you would take to do the task manally. Consider, for example, sorting through a list of about 1000 addresses to extract a few corresponding to a specific ZIP code, or searching through a library card catalog for books authored by Joe Smith on the subject of mating rituals of blue whales. Either of these tasks performed manually would require, even by the best estimates, at least a few days. But a computer can accomplish that in minutes. You can easily understand, therefore, why one of the most common applications of computers is information management.

Information Management or Database Management

Computer-assisted information management begins with organizing data in the form of records. A record is the fundamental unit of a large structure called a database. It contains information relating to one subject. For example, in a library card catalog, each 3x5 index card contains information relating to the bibliography of a book. Each card is, therefore, a record. A collection of related records may be grouped into a file. For example, a collection of employee records constitutes a personnel file. And a list of patients, with their medical history and insurance details, may form a patient file.

From Records to Databases

You can further group files containing related information and create a database. A simple database may only consist of one file. For example, a list of your friends with whom you frequently correspond, their phone numbers, and mailing addresses is such a database. A more complex database, on the other hand, consists of information contained in many files. These databases provide information on a wide variety of subjects to a larger audience. They also facilitate communications between many users for on-line conferences. One such database, CompuServe, provides information on many areas of interest such as science, education, games, programming, etc. It also lets you talk to other computer users and hold conferences within your computer club, over the telephone line using your computer and a modem. You can also shop through their special on-line "vendor service".

Another database popular with investors, is the Dow Jones News and Information Service that provides stock quotes (current and historical), profiles on public corporations and late-breaking news that may affect the performance of stocks. These databases, due to their size and complexity, are handled by more powerful mainframe computers.

Due to the organization and large capacities of databases, many businesses use them to store any information pertaining to their business operations, such as employee payroll files, accounting records, customer files, etc.

A database, therefore, is the next logical step to files in the hierarchy of information organization. It is an organization of a large number of files that may have some inter-related data. A small business, for example, may have a database consisting of personnel files, inventory files, and customer files. As you see, these files are not related to each other. However, they contain information that belongs to one company. In order to access records in any one of these files, you access the database instead of individual files. The database program will, in turn, call the appropriate file in memory and extract the data requested. This type of file organization benefits users handling a large number of files containing related data. Ordinarily, in order to access any information in these files, you either have to remember the file that contains the specific data or worse, search through

the files one at a time. A database will not only provide you access to that particular file, but also other files that may contain related information. All you have to do is specify certain keywords and cross references recognized by the database.

Applications of Database Management

We have mentioned the advantages of computers in handling large amounts of information and their applications in database management, especially for large businesses and organizations in both government and private sectors. However, our primary interest in understanding the principles of database management is its immediate applications in our lives. To that end, we devote the rest of the article to implementation of database management systems (DBMS) on microcomputers such as the Commodore 64.

Before microcomputers became widely available, our record-keeping systems commonly consisted of either a stack of 3x5 index cards or a cabinet full of file folders. Each card or file contained information about one of a group of similar items (a magazine article perhaps, or a customer with whom you correspond) and was filed in ascending numerical or alphabetical order.

There are several limitations to such a filing system. First, the retrieval of specific records from, say, an index card file, can only be made at one level; that is, if the cards containing a mailing list are arranged in alphabetic order by the client's last name, then you can only access them by last name. If you wish to access selected cards by a specific city or zip code, you will have to search the whole card file. This also makes record updating very tedious. Another disadvantage is obvious: you can not, through such a filing system, print out a list of all, or selected, records.

The card index files are most popular in libraries. They get around some of the above limitations by maintaining three index files for each publication (subject, author and title) and using lots of cheap labor (students).

The availability of Database Management Systems (DBMS) on microcomputers has made possible the storing of a wide variety of information on floppy diskettes, thus eliminating the need for 3x5 card files, large filing cabinets and hours of labor. Information in these files may be indexed in many ways, all of them defined by you. Furthermore, you may quickly select and retrieve any records from file at any time. You may also print records in a report format. The fact that these records are maintained on disk files also makes them much faster and easier to update than a card index file. Some sophisticated programs also allow the merging of records with text files. This feature is especially useful for creating personalized form letters; you can merge a standard letter with many addresses and the computer will automatically print out letters, each with the appropriate address. A calculator function, available in a few programs, even allows you to perform arithmetic and logical operations on records containing numeric data.

The applications of database management systems extend much further than the examples cited above. You may store any type of information that you want by creating your own fields (see definition below). The program will let you search through that information, sort it in any order and generate a list of any or all records in that database. For example, let us assume that you have created a database of your customer accounts in the United States. You can now generate a list of your most valued accounts in the entire U.S., or all accounts in a specific area.

Another DBMS special feature mentioned above is the calculator function. Using this feature, you may perform calculations on parts of your records. This is helpful when you wish to update balance owing in your customer accounts. The calculations may be performed on all or selected records.

In addition to maintaining customer accounts and mailing lists, DBMS are also used for other applications such as stock records, inventories, contract records, student records, sales ledgers, invoices, personnel records, etc.

In the next section, we will suggest some important factors to be considered when you select a database management program for your Commodore 64. We will also provide brief reviews of 15 DBMS programs currently available for the C-64.

There are many factors that you might consider, when selecting a particular DBMS for yourself. The most important is its application. Many programs are available, at prices starting at $25-$30, and going to $150 and above. Some of these are general purpose programs, whereas others are specifically designed for one application. A mailing list, for example, is a DBMS designed to maintain a list of names and addresses. Similarly, there are DBMS designed to aid teachers in keeping track of student grades and attendance. If you only need a DBMS program to maintain, for instance, a relatively simple mailing list or inventory file, then the price vs performance ratio may be an important consideration. If, however, you need some sophisticated features in a DBMS, then an advanced DMBS system costing $100 to $150 may be a good investment. In order to help you evaluate these programs, with respect to your applications, we will present a list of criteria that will prove to be useful. These criteria may be used to compare the available programs to determine the price vs performance ratio of each, and perhaps select the one that is the best buy. These criteria are as follows:

Start-Up Options: When you first load and execute a DBMS program, it offers (via the main menu) a variety of options. Using these options, you may configure the system peripherals from within the program. These options may involve printer set-up, DOS commands, screen background/text color changes, etc.

Printer set-up, for example, will let you set up the program for the particular printer that you have (NEC, Diablo, Spinwriter or Centronics-type parallel).

DOS commands refers to accessing the functions and commands of the disk operating system from within the program. This enables you to format a diskette or obtain the disk directory without exiting the program.

Changing screen background/text colors is useful for getting the best contrast between the background screen and the text for improved readability. Although there are up to 256 color combinations available on the Commodore 64 (sixteen background and sixteen text colors), only a few allow optimum readability.

File Structure and Specifications: The efficiency of a particular DBMS in storing and retrieving data from a file depends primarily upon the file structure used. Relative files and random access files provide the fastest data storage and retrieval. Relative files have the added advantage that the record length may be altered. Records that are stored in sequential files, however, are only accessible in the order in which they are stored, so the last record entered into the file will be the last record accessed. In order to implement DBMS functions such as sorts and searches in sequential files, the data has to be loaded into memory completely. This restricts the size of the file (due to limited memory available in the computer) and makes it less versatile. Updating sequential file records is very time-consuming and tedious. Most good quality DBMS, therefore, use relative or random access files to handle record storage.

Specifications refers to the limitations a program imposes on the file and record structures you can create. For example, a program may allow a maximum of twenty-five fields per record and thirty characters per field in each record. If you wanted to use that program to record a mailing list of customers, these specifications may be sufficient. But if you want to store abstracts of magazine articles, you may not have enough space. If you have a specific application in mind, you will find it easier to choose a particular database management system. Otherwise, the more versatile a particular program is, usually, the more favorable it will be with respect to general applicability.

Advanced Data Handling: This refers to features such as "sorts" and "searches" that are available in most programs. The sort feature allows you to arrange the records in your database in a number of ways. You can sort a file in either alphabetic or numeric order. You may use one or more fields to sort your data. For example, if your database consists of mailing addresses, by using the advanced sort feature you may arrange that list in an alphabetical order by customer's last name or by city. You may also want to rearrange the same list in numeric order by the zip code. The search feature lets you look into your data base for specific records. You can define the criteria by using "conditional" statements (such as IF Last Name = Smith OR City = New York) and the program will automatically search for records that match the criteria defined in the conditional statements. Sorts and searches may be performed at one or multiple levels.

Another feature available in most advanced DBMS is the ability to set up calculated fields in your database. This allows you to perform mathematical operations on specified fields of your database file (such as adding tax to the price of a stock item or averaging student grades). In many cases, you can use BASIC's mathematical operators for your formulas for the calculated fields.

Report Generator: A useful function of a DBMS is its ability to generate user-defined reports which may contain a few or all of the records in the file. The reports may be organized as a table or a listing, and the fields may be positioned anywhere on the paper. This flexibility in defining the report format makes a program versatile in its applicability. You can print mailing labels or get a simple listing of a few names and addresses. You can also print selected fields from each record to form a comparison chart. A good database system will support many different types of printer configurations and print formats.

Special Features: In this category, we have included the various features of a program that either add to its performance and applicability or make it outstanding in comparison with other similar programs. For example, a DBMS might feature an integrated word processor and a programmable calculator. This particular package may be of great value if you plan to use your DBMS for creating personalized form letters or keeping track of inventories. Another system may be designed for storing bibliographies. Its use is therefore limited to a specific application.

Glossary

The following terms are most frequently encountered in the manuals of database programs:

Add Fields: Suppose you have created a database of names and addresses of all your employees. Now you want to add another field (date of birth or starting date on the job) to this file. Some programs will not allow you to add a field to the pre-existing file. In this case you will have to start all over again, create a new file and enter all the records. The ability to add fields to a file is a useful feature that very few programs offer. With most DBMS, you must design the file structure very carefully, since you may not be able to add more fields to your file.

Browse: This feature, available in most database management systems, allows you to look at the records in a file sequentially, starting with the first one. This 'browsing' may also be performed in a reverse order (i.e., if you are currently working with the 100th record and you want to view the preceding few records, then you may browse in a descending order).

Calculator: This is a useful feature in a database management system. It lets you perform arithmetic calculations on the numeric data types in your records. The types of calculations that you can perform vary from program to program. The feature, however, adds to the versatility of the DBMS in its applications.

Conditional Statement: This statement consists of logical operators (such as IF, THEN, GE, LE, EQ) that may be used to select specific records from a database. For example, you may use a statement like, "IF last name EQ Smith AND City EQ San Francisco," to tell the computer to select only records which have "Smith" in the last name field and "San Francisco" in the city field.

Database: A database is a collection of information organized in the form of records. It may be a list of customers' phones and addresses or an inventory of items in stock. The term database is also commonly used in relation to large information networks such as the Source or the Dow Jones News Retrieval service.

Database Management System: Often abbreviated DBMS; refers to a collection of computer programs that facilitate the creation and use of a database in the form of a file(s). It is an electronic filing system that offers efficiency in data storage and retrieval.

Editing: Once you have entered your records in a file and want to update a specific record or you entered some information incorrectly and want to correct it, you will be working in the editing mode of a DBMS. The manner in which this mode is implemented in each program and the efficiency with which your are able to update your records is considered here.

Field: A field is a specific data type. This may be the book title in a library card catalog, or a zip code in a mailing list. A record consists of many inter-related fields. Information within a field may or may not be identical between records. For example, in a file containing a mailing list, each record might contain five fields: name, street, city, state and zip code of clients. If two clients live in the same block, their zip code will be the same; therefore, the information in those fields will be identical in the respective records. Fields may be used to perform sorts and searches within a file, although some DBMS will search only on key fields.

File: A collection of records on disk that are saved under a unique name. The records consist of identical data types (fields). For example, you may have a file containing a list of vendors that manufacture software for the Commodore 64. All records in this file will have identical field structure (i.e., all records will contain the name and address of the vendor and the product that they manufacture).

Function Keys: The Commodore 64 has four undefined, programmable keys on the right side of the keyboard. These keys are often referred to as the function keys. By using these keys in conjunction with the shift key, you can actually perform up to 8 functions in your program. The function keys add to the ease and efficiency of using the features of a certain program. For example, you can use the function keys to select various menu options of a program. Without them, you would normally have to physically type in each option.

Help Screen: Some programs display a list of commands and functions to help you select the right command for a specific function. This way, you are not forced to memorize all the commands of the system and their specific functions. This is referred to as a help screen.

Key: A key is an identifier consisting of one or more fields. It is used to sort, search and format output of desired data elements. If you have a database of sales records, for example, then you may identify one or more fields (salesman, product name, etc) in this database as key fields. This will enable you to sort or search the entire database for specific records by establishing criteria using these key fields. Some DBMS will allow searches only on key fields, while others will search for non-key fields with a slower process.

Menu-Driven: Many DBMS packages display the master menu when the program is first executed. Selection of a function or option from the master menu results in the display of another menu that contains more detailed features of that particular function. Such a system or program is called menu-driven.

On Screen Prompts: These are the prompts that a program displays on the monitor screen every time it requires you to perform a certain task such as inserting a new disk or change printer paper, etc. These prompts are very helpful since they do not require you to memorize every step that you go through during the program execution. The program keeps you informed of the next step and any inputs that it needs from you.

Random Access File: A type of disk file that allows you to directly access records through the program by specifying the drive, track and sector number. These files are not given names and do not appear in the disk directory.

Record: A record is a collection of data items (fields). In a personnel file, for example, the information on each employee is considered a record. The maximum number of records that a database may contain is limited by the size of each record and available space on a diskette.

Relative Files: These are similar to random files, except that the files are given unique identifiers (file names) and the record length in the file is alterable. A relative file may contain up to 720 records.

Report: A report is a user-defined printout (or hard copy) of selected information in the database. Many different kinds of reports may be generated by a DBMS such as lists, forms, tables, etc.

Search: This function allows you to search a file for specific records that you have defined with conditional statements.

Security: A feature available in some DBMS that allows you to restrict access to those with a password.

Sequential Files: A file that sequentially stores data on disk. Access to data is made in the same order each time (from the first element in the file to the last element in the file). This type of structure makes searches or record updating very time-consuming and tedious.

Sort: This is a function (available in some DBMS) that allows you to rearrange the records in your database alphabetically or numerically. You may also be able to select (using conditional statements) the records to be sorted instead of sorting the whole file.

Spreadsheet Features: Some DBMS have built-in features which allow you to build a spreadsheet from selected database records.

Start-Up Options: These are the options available to you through the program when you first load and execute it. The options are displayed through the main menu. They may include printer set-up, disk initialization, color adjustment, etc.

Word Processor Interfacing: Some DBMS allow you to create a file from the database which can be further processed by a word processor.

Next Month

Part II of this article will deal with specific database management systems available for the Commodore 64.

Acknowledgement

Some of the material in this article is based on work done by the author for:

The Commodore 64 Buyer's Guide, by Gary Phillips, Terry Silveria and Sanjiva Nath, published by the R. I. Brady Publishing Co., July 1984.

BASIC/ML Data Transfer

by Mark 'Jay' Johanson
Germantown, Ohio

A Number of Techniques are Presented to Transfer Data Between BASIC Programs and Machine Language Subroutines

While writing programs using a BASIC interpreter is very easy and convenient (or at least, easier and more convenient than most other methods), I am sure that I am not alone in the discovery that BASIC programs sometimes run extremely slow. The obvious alternative is to use machine language, but writing in machine language, even with the aid of an assembler, is significantly more difficult than writing in BASIC; sometimes it can become overwhelming.

A common solution to this dilemma is to write machine-language subroutines into your BASIC program, using BASIC for the bulk of the program because of its convenience, but using machine language for a few critical routines for the sake of speed, choosing routines for which ML will give significant performance improvements. While such a scheme can be very effective, there are several problems (opportunities?) which must be overcome. It is my purpose here to address one particular problem involved in such programs: transferring data between the two languages, specifically on the Commodore Vic-20 and 64. While users of other machines may be able to make use of some of the basic principles that I will discuss here, many of the details are, unfortunately, tied to the workings of Commodore's BASIC interpreter and operating system.

In the discussion which follows I assume that the reader has some knowledge of 6502 machine language. As an aid to comprehension, in all my sample programs, I include the assembler equivalent of any machine language code as REMarks following the POKE values. As it is not my purpose here to discuss the question of where to locate an ML routine in memory, in these examples I will simply put my ML routines (and data) in the cassette buffer.

POKEing Along

The most obvious way to make BASIC's data available to machine language is via the POKE statement and, likewise, data can be retrieved by BASIC with a PEEK function. If you have been using machine language routines you are probably familiar with this technique, so I will only discuss it briefly here.

In order to use this method, it is only necessary that you decide on some specific memory location which is to hold the data and then cause both languages to access it. Program 1 demonstrates this by reading in a number using BASIC, passing it to a machine language routine which adds two to it and then using BASIC again to print out the sum. (And before you point out that this is a totally inane use of machine language, let me hasten to add that it is by no means intended to be a useful program: it is just a demonstration.)

Remember that POKE values are limited to one byte, i.e. 0 thru 255. A Commodore BASIC integer is two bytes, so to allow for a full range of integer values we must use something more like Program 2. This begins to illustrate some of the problems with this method of transferring data: if we want to pass more than one byte things begin to get rather involved.

This method has the advantage of being straightforward and general, but as the amount of data to be passed becomes large, all the PEEKs and POKEs can become very tedious. Furthermore, the POKE is one of the slowest instructions on the Commodore, so if you have a lot of them, they can slow your program down.

Using USR

A second method of transferring data involves the USR function. This function is seldom used but can be very handy in certain situations. Before using it you must first POKE the address of the machine language routine into locations 1 and 2 on the Vic-20, locations 785 and 786 on the C-64, in the standard low-high format, i.e.

```
POKE 1,AD AND 255
POKE 2,INT(AD/256)
```

(for the Vic). It can then be used just like any other function, ranging from a simple A=USR(B) to including it within a complex expression and, just as for any other function, the value within the parentheses may itself be an expression.

When the function is used within a BASIC statement, control will be passed to your machine language routine in a manner similar to what happens when you use a SYS statement, and control will be returned to BASIC when your routine executes an RTS instruction (or more correctly, when it executes one more RTS instruction than it has JSR instructions). But when the ML routine begins, BASIC will have placed the value found within the parentheses (the result of the computation if this was an expression) as a floating point number in locations 97 thru 101. When your routine finishes, it should place a floating-point value in this same location: this number will be used as the result of the function. For example, if you had the BASIC statement:

```
X=2*USR(A/B+2)-7
```

then, when your ML routine was called, the value in locations 97-101 would be the result of the computation A/B + 2. If your routine deposited the number 5 into this location, then X would end up being assigned the value 2*5-7, or 3.

If you wish, you may work directly with these floating-point values. Unfortunately, floating-point numbers are very difficult to work with — personally, aside from a couple of demonstration routines to prove to myself that I could do it, I have never used them. However, Commodore has graciously provided us with conversion routines: a floating-to-fixed routine at location 53674, and fixed-to-floating at 54161. Conveniently, both these routines use 97 thru 101 as the location for the floating-point number, the same location accessed by USR. They use the A and Y registers for the integer value, with the most significant byte in A and the least significant in Y. (According to the Vic 20 manual, they use memory locations 20 and 21 for the integer value. Unfortunately, this does not appear to be the case. Numbers do appear there whenever BASIC does an integer conversion, but these locations are not accessed by the routines mentioned above. Perhaps there is some other routine which moves values between A:Y and 20-21 but, as this is a relatively trivial operation, I have not bothered to look for such a routine within the operating system.)

Thus, all the ML routine must do upon execution is execute the floating-to-fixed conversion with a JSR 53674, do whatever work it desires with the integer value which will now be in A:Y and then, when it is finished, put the desired return value in A:Y and execute the fixed-to-floating routine with a JSR 54161. This is demonstrated in Program 3 which, again, will simply add two to the entered number.

The major advantage of the USR function is that it makes your BASIC program simpler, faster and more readable. It is not necessary to do cumbersome PEEKs and POKEs to move the data around and the resultant value of the function can be used directly in a more complex formula without any intermediate steps. It does require two extra instructions in the ML routine — the JSR's to do the conversions — but this is a small penalty and, for that matter, these routines end up putting the value into registers, which you would probably have taken a couple of instructions to do anyway. A bigger drawback is that you can only get one value into the function and if you want to use more than one USR routine in the same program you may end up having to continually alter which one is 'active' by POKEing values into the USR vector; that destroys the advantage of not having to do POKEs to get the data in. In short, don't try to use USR for every ML routine you ever write from now on; USR is only of value in certain limited situations. But when it is helpful, it can make your program much more elegant and slightly faster.

Make-Believe Registers

I stumbled upon a third method of transferring data between BASIC and ML almost by accident. I only recall seeing it used by someone else once, and in that instance the writer included it in a program without explanation.

In the memory maps found in the Vic-20 and C-64 manuals, for locations 780 thru 783 one finds the cryptic notes, 'storage for 6502 A register', 'storage for 6502 X register', etc. I found no further explanation in the manual of what these are for, so one day I became curious about them and tried some experimenting which led me to discover this useful fact: whenever you use a SYS statement in a BASIC program, before control is transferred to your routine the system loads the registers with the values found at locations 780-783; when your routine exits, before control is returned to BASIC the system stores the values of the registers at these locations.

At first glance, this feature may seem to be of only marginal value. Instead of saying POKE 828,N before calling your ML routine and then having the ML routine begin with a LDA 828, you could simply say POKE 780,N and then when your routine began the desired value would be waiting in the A register. So big deal, we've saved one instruction. A somewhat more useful application would be for an ML routine to leave data in the registers; the next time this routine is executed the registers will appear to have been unchanged by anything BASIC may have done in the meantime, because the system will have saved off the registers when it finished and then restored them with the same values when the routine was re-entered (assuming neither the BASIC program nor some other ML routine had modified locations 780-783). Still, this would only save us from having to do some POKEs.

However, there is one situation where this feature can be quite useful, namely, when we want to use one of the kernal routines from within a BASIC program. This is best illustrated with an example. Commodore BASIC includes no cursor positioning command. Thus, if you want to plant the cursor at a specific location on the screen using BASIC, about the best you can do is something like Program 4, using 'home' followed by variable numbers of 'down's and 'right's to get the cursor to the desired location. This example creates a string with the maximum number of cursor movement keys you can use (on the Vic-20), and then does LEFT$'s on them to get the desired number. For demonstration purposes, it simply asks for a row and column number, prints an asterisk at that location and then waits for any key to be struck to tell it to clear the screen and repeat the process (indefinitely).

But this is inefficient and inelegant. Hope appears when we note that the kernal does have a cursor positioning routine, beginning at location 65520. However, this expects the row and column to be in the X and Y registers respectively, and BASIC cannot directly modify the contents of the registers. (The BASIC interpreter is using the registers constantly while executing our BASIC program, so even if BASIC did include an instruction that modified a register, the inserted value would quickly be overwritten.) Thus, it would appear that we are forced to pass the desired row and column to an ML routine which will actually load the registers and execute the call. An example of this is given as Program 5 which performs the same task as Program 4, but using the kernal plot routine instead of strings of cursor control characters. Note that, even though the ML routine is trivial, the program still has to go to a certain amount of trouble to load and execute it. Perhaps this is not a terribly heavy price, but there is a better way.

Program 6 uses the location 780-783 feature to call the kernal plot routine from BASIC without the need for any 'ML interface routine'. It performs the same function as Programs 4 and 5, but note that actually doing the plot takes only four statements: three POKE's and a SYS. (The third POKE is needed because the plot routine can actually perform two functions: planting the

cursor at a given location, or telling you where the cursor is currently sitting. It decides which function to perform depending on the contents of the carry flag, which we here set to zero by means of the third POKE. This will also set all the other flags to zero, but, as we don't care about their values, it's easiest to set them all to zero and avoid any possible confusion.)

As you can see, while it is unlikely that this facility would be of any help in passing data into your own routines, it can greatly simplify the use of routines from the kernal. By the way, note that all of this works with the SYS statement; it does not work with USR.

BASIC's Backyard

Someone might reasonably ask, "Why must we make a copy of the data to pass to an ML routine? Why not let the ML routine use BASIC's variables directly, in the same locations in which BASIC actually stores them for its own use?" This thought leads to a technique which is more complex than those I have discussed previously, but which is extremely efficient, especially when there is a great deal of data to be passed.

The obvious hurdle to be overcome here is finding where BASIC stores its variables. We could investigate where the variable table is located and how it is laid out and then develop a routine to find any desired variable, but this is not necessary. BASIC has to do that work itself all the time, so we can simply let it do this for us.

If you look at a memory map you will see that locations 71 and 72 are described as 'current variable address'. Using this clue I experimented a bit and discovered that these two bytes always contain the address of the last variable that you have used in your program. Thus, if you code a line such as N=0: POKE 251,PEEK(71): POKE 252,PEEK(72), you will put the address of N into locations 251-252. (Note that if you entered N=0: A1=PEEK(71): A2=PEEK(72) you would not end up with the address of N in A1 and A2, because by using two more variables you will have overlaid the previous address. You must avoid using any other variables until you have copied the address into a safe place. This essentially means that you must POKE it somewhere, as that's the only way (that I can think of) to move data without using a variable.) If you execute a statement such as this at the beginning of a BASIC program and stow the variable address in some convenient location, an ML routine could refer to that variable from then on. For an elementary variable, it is not necessary to redo the look-up as Commodore's BASIC will never move a variable once it has been created (an array, however, may be moved).

The next issue to be considered is exactly what you will find at this address. This depends on the variable type. If it is a floating-point number, at the given address will be the five byte floating-point value. As I mentioned earlier, floating-point numbers are difficult to work with using machine language and so I will dispense with any further discussion of them here.

More useful are integer variables. For these the address is that of a two-byte integer, with the most significant byte stored first, followed by the least significant byte. Note that this is the reverse of the order normally used on the 6502. (I've forgotten this and slipped up several times.)

Program 7 uses this technique to perform the same dull 'add 2' operation. A few points are worth comment here. The most straightforward way to use the variable address is to store it somewhere on page zero and then use the (addr), Y addressing mode to access the data. For this example I have put it at locations 251 and 252, two of the four page zero locations which Commodore promises that BASIC will never disturb. Four bytes is only enough room to store two variable addresses permanently, so in real life you would probably have to store the address elsewhere and then move it to page zero when it is needed. This is demonstrated in Program 7B.

While the work that must be done in BASIC is no more involved than that required for any other method, the ML routine does need several extra instructions to find the variable. If your routine is short and only uses a few bytes of data, this extra work is probably not worth it. As your routine becomes larger a few extra instructions become less significant (as a percentage), and, if you must pass a lot of data back and forth, this technique lets the ML routine do all the work rather than BASIC, which is a much more efficient system.

Speaking of the amount of data to be passed, consider the following: An integer occupies two bytes and a floating-point number 5, but a string variable may take up to 255 bytes. To try to pass this much data to an ML routine by copying it with POKEs would be extremely cumbersome; this is where you would save the most by working on the variable directly. Program 8 demonstrates this with a routine to examine a string and replace every occurrence of a dollar sign with a pound sign. (Which is about as useful as reading in a number and adding two to it.) For string variables, the address found in locations 71-72 points to a three byte area containing first a one-byte length value and then a two-byte address of the actual string. Thus we must follow two levels of indirection to get to the actual data; the first address points us to an area containing, not the data itself, but rather another address pointer to follow. Note that while the address of this three-byte area will never change during the execution of a program, the second address, the address of the string itself, will change everytime BASIC modifies the string, as well as on other occasions when BASIC does its 'garbage collection' to clean up unused areas in string space. So even if you don't modify the string, don't count on it staying put. Have the ML routine reload this second address every time it executes.

There is one caution to be borne in mind when modifying string variables in place; while you may freely change the contents of any byte in the string and you may make the string shorter by altering the length value, you should definitely avoid trying to make the string longer, as you probably have no idea what may happen to be sitting in the space following the present contents of the string. Usually this will be another string variable and you normally don't want to destroy other variables. While it is possible in principle to create a dummy variable from which you will take the needed space, you would have to be careful that the garbage collection routine did not get invoked at the wrong time and move your variables in relation to each other. I have found it far more practical to let BASIC either do all the lengthening (by concatenating strings together) or to add a bunch of dummy characters to the end of the string so that the ML routine need only shorten it by the number of added bytes that it decides it doesn't need. For example, before executing the ML routine, use BASIC to add 10 spaces to the end of the string. If the ML routine then decides that six extra spaces were needed, it reduces the length by the difference, or four bytes.

I tend to prefer this 'in-place' method of data transfer because of its 'cleanness'; the BASIC program isn't cluttered up with a lot of POKEs, but simply sets variables just like it would before a GOSUB. But if only a couple of bytes of data are being passed, or if the ML routine is rather short, then the extra ML code required seems excessive.

Parting Thoughts

Each of the four methods of transferring data which I have presented here has its own uses. The simple PEEK/POKE is good for small amounts of data and general 'quick-and-dirty' applications; USR is handy when you want an ML routine to produce a result which will be used in an expression, or when the input to it is the result of an expression; the register storage area is convenient for setting up calls to kernal routines; and working directly in BASIC's variable area is a help when there is a large amount of data to be passed, especially string variables.

I don't doubt that other techniques could be found with their own particular advantages. It's good to have a variety of techniques at your disposal — just because something works well in one situation, don't assume that that is all you'll ever need. A monkey wrench is a handy tool: it can be used on bolts of almost any size and in a pinch you can use it as a hammer or a crowbar. But the job will be a lot easier if you take something in your toolbox besides a monkey wrench.

BML 1

```
10 REM PROGRAM 1
20 REM TRANSFER DATA WITH PEEKS AND POKES
30 FOR AD=840 TO 849:READ B:POKE AD,B:NEXT
110 DATA 173,60,3:REM LDA 828
120 DATA 24:REM CLC
130 DATA 105,2:REM ADC #2
140 DATA 141,60,3:REM STA 828
150 DATA 96:REM RTS
200 REM
210 INPUT N
220 POKE 828,N
230 SYS 840
240 PRINT PEEK(828)
```

BML 2

```
10 REM PROGRAM 2
20 REM TRANSFER TWO BYTES WITH PEEKS AND POKES
30 FOR AD=840 TO 857:READ B:POKE AD,B:NEXT
110 DATA 173,60,3:REM LDA 828
120 DATA 24:REM CLC
130 DATA 105,2:REM ADC #2
140 DATA 141,60,3:REM STA 828
150 DATA 173,61,3:REM LDA 829
160 DATA 105,0:REM ADC #0
170 DATA 141,61,3:REM STA 828
190 DATA 96:REM RTS
200 REM
210 INPUT N
220 POKE 828,N AND 255:POKE 829,INT(N/256)
230 SYS 840
240 PRINT PEEK(828)+256*PEEK(829)
```

BML 3

```
10 REM PROGRAM 3
20 REM TRANSFER DATA VIA USR FACILITY
30 FOR AD=840 TO 875:READ B:POKE AD,B:NEXT
101 DATA 32,170,209:REM JSR FIXFL
102 DATA 140,60,3:REM STY 828
103 DATA 141,61,3:REM STA 829
110 DATA 173,60,3:REM LDA 828
120 DATA 24:REM CLC
130 DATA 105,2:REM ADC #2
140 DATA 141,60,3:REM STA 828
150 DATA 173,61,3:REM LDA 829
160 DATA 105,0:REM ADC #0
170 DATA 141,61,3:REM STA 829
171 DATA 172,60,3:REM LDY 828
172 DATA 173,61,3:REM LDA 829
180 DATA 32,145,211:REM JSR FLFIX
190 DATA 96:REM RTS
200 REM
205 POKE 1,72:POKE 2,3
210 INPUT N
220 N2=USR(N)
240 PRINT N2
```

BML 4

```
10 REM PROGRAM 4
20 REM CURSOR POSITIOING WITH DOWN'S AND RIGHT'S
30 R$="{HOME,DOWN22}"
40 C$="{RIGHT22}"
100 PRINT"{CLEAR}";:INPUT"ROW,COLUMN";R,C
110 PRINT LEFT$(R$,R);LEFT$(C$,C-1);"*";
120 GET I$:IF I$="" THEN 120
130 GOTO 100
```

BML 5

```
10 REM PROGRAM 5
20 REM CURSOR POSITIOING WITH KERNAL PLOT ROUTINE
30 FOR AD=840TO850:READ B:POKE AD,B:NEXT
40 DATA 174,60,3:REM LDX 828
50 DATA 172,61,3:REM LDY 829
60 DATA 24:REM CLC
70 DATA 32,240,255:REM JSR PLOT
80 DATA 96:REM RTS
100 PRINT"{CLEAR}";:INPUT"ROW,COLUMN";R,C
110 POKE 828,R-1:POKE 829,C-1:SYS 840:PRINT"*";
120 GET I$:IF I$="" THEN 120
130 GOTO 100
```

BML 6

```
10 REM PROGRAM 6
20 REM CURSOR POSITIOING WITH KERNAL PLOT ROUTINE
   AND NO ML INTERFACE
100 PRINT"{CLEAR}";:INPUT"ROW,COLUMN";R,C
110 POKE 781,R-1:POKE 782,C-1:POKE 783,0:
    SYS 65520:PRINT"*";
120 GET I$:IF I$="" THEN 120
130 GOTO 100
```

BML 7

```
10 REM PROGRAM 7
20 REM TRANSFER DATA BY VARIABLE ADDRESS
30 FOR AD=840 TO 856:READ B:POKE AD,B:NEXT
40 DATA 160,1:REM LDY #1
50 DATA 24:REM CLC
60 DATA 177,251:REM LDA (251),Y
70 DATA 105,2:REM ADC #2
80 DATA 145,251:REM STA (251),Y
90 DATA 136:REM DEY
100 DATA 177,251:REM LDA (251),Y
110 DATA 105,0:REM ADC #0
120 DATA 145,251:REM STA (251),Y
130 DATA 96:REM RTS
200 REM GET ADDRESS
210 N%=0:POKE 251,PEEK(71):POKE 252,PEEK(72)
300 REM DO IT
310 INPUT N%
320 SYS 840
330 PRINT N%
```

BML 7B

```
10 REM PROGRAM 7B
20 REM TRANSFER DATA BY VARIABLE ADDRESS--
   NOT RELYING ON PAGE 0 SPACE
30 FOR AD=840 TO 866:READ B:POKE AD,B:NEXT
40 DATA 173,60,3:REM LDA 828
50 DATA 133,251:REM STA 251
60 DATA 173,61,3:REM LDA 829
70 DATA 133,252:REM STA 252
80 DATA 160,1:REM LDY #1
90 DATA 24:REM CLC
100 DATA 177,251:REM LDA (251),Y
110 DATA 105,2:REM ADC #2
120 DATA 145,251:REM STA (251),Y
130 DATA 136:REM DEY
140 DATA 177,251:REM LDA (251),Y
150 DATA 105,0:REM ADC #0
160 DATA 145,251:REM STA (251),Y
170 DATA 96:REM RTS
200 REM GET ADDRESS
210 N%=0:POKE 828,PEEK(71):POKE 829,PEEK(72)
300 REM DO IT
310 INPUT N%
320 SYS 840
330 PRINT N%
```

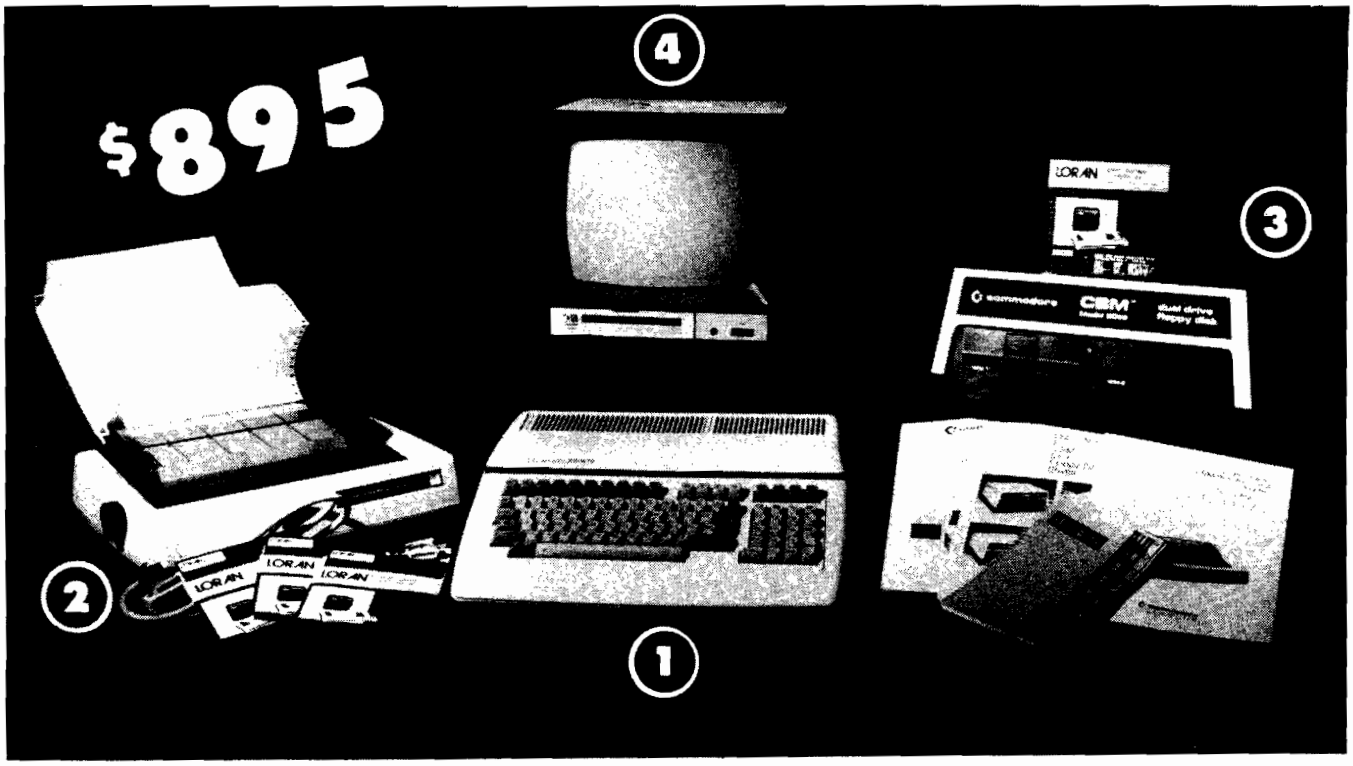
BML 8

```
10 REM PROGRAM 8
20 REM TRANSFER DATA BY VARIABLE ADDRESS--STRINGS
30 FOR AD=840 TO 874:READ B:POKE AD,B:NEXT
40 DATA 160,1:REM LDY #1
45 DATA 177,251:REM LDA (251),Y
50 DATA 133,253:REM STA 253
55 DATA 200:REM INY
60 DATA 177,251:REM LDA (251),Y
65 DATA 133,254:REM STA 254
70 DATA 160,0:REM LDY #0
75 DATA 177,251:REM LDA (251),Y
80 DATA 141,60,3:REM STA 828
85 DATA 177,253:REM LDA (253),Y
90 DATA 201,36:REM CMP #'$'
95 DATA 208,4:REM BNE +4
100 DATA 169,92:REM LDA #'{POUND}'
105 DATA 145,253:REM STA (253),Y
110 DATA 200:REM INY
115 DATA 204,60,3:REM CPY 828
120 DATA 48,240:REM BMI -16
130 DATA 96:REM RTS
200 REM GET ADDRESS
210 N$="":POKE 251,PEEK(71):POKE 252,PEEK(72)
300 REM DO IT
310 INPUT N$
320 SYS 840
330 PRINT N$
```

NEW 128K —MEGA BYTE DUAL DISK DRIVE—80 COLUMN

COMPUTER SYSTEM SALE!

HOME • BUSINESS • WORD PROCESSING



LOOK AT ALL YOU GET FOR ONLY **$895.**

	LIST PRICE
① B128 COMMODORE 128K 80 COLUMN COMPUTER	$ 995.00
② 4023 - 100 CPS - 80 COLUMN BIDIRECTIONAL PRINTER	499.00
③ 8050 DUAL DISK DRIVE (over 1 million bytes)	1795.00
④ 12" HI RESOLUTION 80 COLUMN MONITOR	249.00
• BOX OF 10 LORAN LIFETIME GUARANTEED DISKS	49.95
• 1100 SHEETS FANFOLD PAPER	19.95
• ALL CABLES NEEDED FOR INTERFACING	102.05
TOTAL LIST PRICE	**$3717.95**

$895 SALE PRICE

PLUS YOU CAN ORDER THESE BUSINESS PROGRAMS AT SALE PRICES

	LIST	SALE
Professional 80 Column Word Processor	$149.95	**$99.00**
Professional Data Base	$149.95	**$99.00**
Accounts Receivable	$149.95	**$99.00**
Accounts Payable	$149.95	**$99.00**
Payroll	$149.95	**$99.00**
Inventory	$149.95	**$99.00**
General Ledger	$149.95	**$99.00**
Financial Spread Sheet	$149.95	**$99.00**

PRINTER REPLACEMENT OPTIONS

(replace the 4023 with the following at these sale prices)

	LIST	SALE
Olympia Executive Letter Quality Serial Printer	$699.00	**$399.00**
Comstar Hi-Speed 160 CPS 15½" Serial Business Printer	$779.00	**$499.00**
Telecommunications Deluxe Modem Package	$199.00	**$139.00**

15 DAY FREE TRIAL. We give you 15 days to try out this SUPER SYSTEM PACKAGE!! If it doesn't meet your expectations, just send it back to us prepaid and we will refund your purchase price!!
90 DAY IMMEDIATE REPLACEMENT WARRANTY. If any of the SUPER SYSTEM PACKAGE equipment or programs fail due to faulty workmanship or material we will replace it IMMEDIATELY at no charge!!

Add $50.00 for shipping and handling!!
$100.00 for Alaska and Hawaii orders.
WE DO NOT EXPORT TO OTHER COUNTRIES
Enclose Cashiers Check, Money Order or Personal Check. Allow 14 days for delivery. 2 to 7 days for phone orders. 1 day express mail! We accept Visa and MasterCard. We ship C.O.D. to continental U.S. addresses only.

PROTECTO ENTERPRIZES WE LOVE OUR CUSTOMERS.
BOX 550, BARRINGTON, ILLINOIS 60010
Phone 312/382-5244 to order

FLOPPY DISKS SALE *98¢ ea.

Economy Model or Cadillac Quality

LORAN CERTIFIED PERSONAL COMPUTER DISK ***We have the lowest prices!*** LORAN CERTIFIED PERSONAL COMPUTER DISK

***ECONOMY DISKS**

Good quality 5¼" single sided single density with hub rings.

Bulk Pac	100 Qty.	98¢ ea.	Total Price	$98.00
	10 Qty.	$1.20 ea.	Total Price	12.00

CADILLAC QUALITY (double density)

• Each disk certified • Free replacement lifetime warranty • Automatic dust remover

For those who want cadillac quality we have the Loran Floppy Disk. Used by professionals because they can rely on Loran Disks to store important data and programs without fear of loss! Each Loran disk is 100% certified (an exclusive process) plus each disk carries an exclusive FREE REPLACEMENT LIFETIME WARRANTY. With Loran disks you can have the peace of mind without the frustration of program loss after hours spent in program development.

100% CERTIFICATION TEST

Some floppy disk manufacturers only sample test on a batch basis the disks they sell, and then claim they are certified. Each Loran disk is individually checked so you will never experience data or program loss during your lifetime!

FREE REPLACEMENT LIFETIME WARRANTY

We are so sure of Loran Disks that we give you a free replacement warranty against failure to perform due to faulty materials or workmanship for as long as you own your Loran disk.

AUTOMATIC DUST REMOVER

Just like a record needle, disk drive heads must travel hundreds of miles over disk surfaces. Unlike other floppy disks the Loran smooth surface finish saves disk drive head wear during the life of the disk. (A rough surface will grind your disk drive head like sandpaper). The lint free automatic CLEANING LINER makes sure the disk-killers (dust & dirt) are being constantly cleaned while the disk is being operated. PLUS the Loran Disk has the highest probability rate of any other disk in the industry for storing and retaining data without loss for the life of the disk.

Loran is definitely the Cadillac disk in the world

Just to prove it even further, we are offering these super LOW INTRODUCTORY PRICES

List $4.99 ea. INTRODUCTORY SALE PRICE $2.99 ea. (Box of 10 only) Total price $29.90

$3.33 ea. (3 quantity) Total price $9.99

All LORAN disks come with hub rings and sleeves in an attractive package.

DISK DRIVE CLEANER $19.95

Everyone needs a disk drive doctor

(Coupon Price $16.95)

FACTS

- 60% of all drive downtime is directly related to poorly maintained drives.
- Drives should be cleaned each week regardless of use.
- Drives are sensitive to smoke, dust and all micro particles.
- Systematic operator performed maintenance is the best way of ensuring error free use of your computer system.

The Cheetah disk drive cleaner can be used with single or double sided 5¼" disk drives. The Cheetah is an easy to use fast method of maintaining efficient floppy diskette drive operation.

The Cheetah cleaner comes with 2 disks and is packed in a protective plastic folder to prevent contamination.

List $29.95 / Sale $19.95 * Coupon $16.95

Add $3.00 for shipping, handling and insurance. Illinois residents please add 6% tax. Add $6.00 for CANADA, PUERTC RICO, HAWAII, ALASKA, APO-FPO orders. Canadian orders must be in U.S. dollars. WE DO NOT EXPORT TO OTHER COUNTRIES.

Enclose Cashiers Check, Money Order or Personal Check. Allow 14 days for delivery, 2 to 7 days for phone orders, 1 day express mail!

VISA — MASTER CARD — C.O.D.

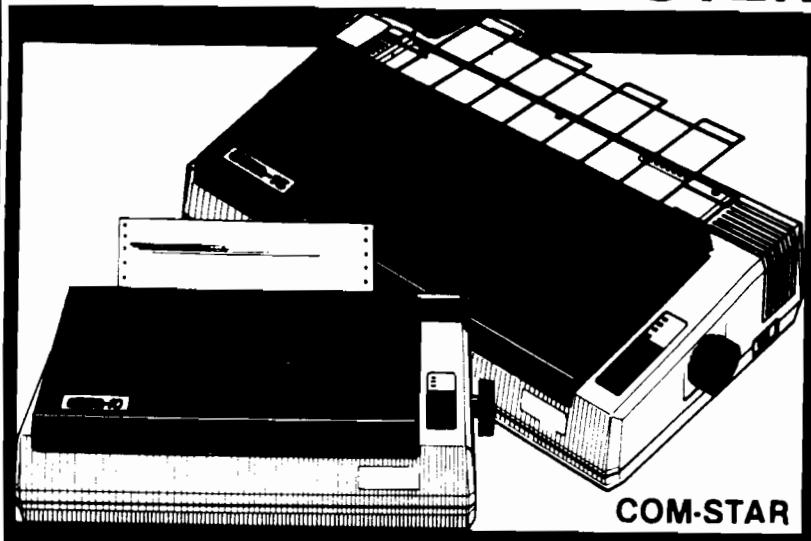
No C.O.D. to Canada, APO-FPO.

PROTECTO ENTERPRIZES WE LOVE OUR CUSTOMERS!

BOX 550, BARRINGTON, ILLINOIS 60010

Phone 312/382-5244 to order

FANTASTIC COMPUTER PRINTER SALE!!!



COM-STAR T/F

Tractor
Friction
Printer

only $ 169**

- Lowest Priced, Best Quality, Tractor-Friction Printers in the U.S.A.
- Fast 80-120-160 Characters Per Second • 40, 46, 66, 80, 96, 132 Characters Per Line Spacing
- Word Processing • Print Labels, Letters, Graphs and Tables • List Your Programs
- Print Out Data from Modem Services • "The Most Important Accessory for Your Computer"

** DELUXE COMSTAR T/F 80 CPS Printer — $169.00

This COMSTAR T/F (Tractor Friction) PRINTER is exceptionally versatile. It prints 8½" x 11" standard size single sheet stationary or continuous feed computer paper. Bi-directional, impact dot matrix 80 CPS, 224 characters. (Centronics Parallel Interface).

Premium Quality 120-140 CPS 10" COM-STAR PLUS+ Printer $249.00

The COM-STAR PLUS+ gives you all the features of the COMSTAR T/F PRINTER plus a 10" carriage, 120-140 CPS, 9 x 9 dot matrix with double strike capability for 18 x 18 dot matrix (near letter quality), high resolution bit image (120 x 144 dot matrix), underlining, back spacing, left and right margin settings, true lower decenders with super and subscripts, prints standard, italic, block graphics and special characters. It gives you print quality and features found on printers costing twice as much!! (Centronics Parallel Interface) (Better than Epson FX80). List $499.00 **SALE $249.00**

Premium Quality 120-140 CPS 15½" COM-STAR PLUS+ Business Printer $349.00

Has all the features of the 10" COM-STAR PLUS+ PRINTER plus 15½" carriage and more powerful electronics components to handle large ledger business forms! (Better than Epson FX 100). List $599

SALE $349.00

Superior Quality 140-160 CPS 10" COM-STAR PLUS+ IBM IBM Pers/Bus Printer $369.00

Has all the features of the 10" COM-STAR PLUS+ PRINTER! It is especially designed for all IBM personal computers! 140-160 CPS HIGH SPEED PRINTING 100% duty cycle, 2K buffer, diverse character fonts, special symbols and true decenders, vertical and horizontal tabs.

A RED HOT IBM personal business printer at an unbelieveable low price of $369.00 (centronics parallel interface)

List $699 **SALE $369.00**

Superior Quality 160-180 CPS 10" COM-STAR PLUS+ HS Business Printer $369.00

The **Super Com-Star+ High Speed Business Printer** 160-180 CPS has a 10" carriage with all the Com-Star+ features built in! The 15½" **High Speed Business Printer** is especially designed with more powerful electronics to handle larger ledger business forms! Exclusive bottom feed! (Centronics parallel interface)

15½" printer is also compatable with IBM Personal/Business Computers! 15½" Printer List $799.00 ***SALE* $469.00**

Olympia

Executive Letter Quality DAISY WHEEL PRINTER $379.00

This is the worlds finest daisy wheel printer **Fantastic Letter Quality**, up to 20 CPS bidirectional, will handle 14.4" forms width! Has a 256 character print buffer, special print enhancements, built in tractor-feed (Centronics Parallel and RS232C Interface) List $699 **SALE $379.**

• *15 Day Free Trial - 1 Year Immediate Replacement Warranty*

PARALLEL INTERFACES

For VIC-20 and COM-64 — $49.00 For Apple computers — $79.00 Atari 850 Interface — $79.00 For ALL IBM Computers — $89.00

Add $14.50 for shipping, handling and insurance. Illinois residents please add 6% tax. Add $29.00 for CANADA, PUERTO RICO, HAWAII, ALASKA, APO-FPO orders. Canadian orders must be in U.S. dollars. WE DO NOT EXPORT TO OTHER COUNTRIES.

Enclose Cashiers Check, Money Order or Personal Check. Allow 14 days for delivery, 2 to 7 days for phone orders, 1 day express mail!

VISA—MASTER CARD—We Ship C.O.D. to U.S. Addresses Only

PROTECTO ENTERPRIZES (WE LOVE OUR CUSTOMERS)

BOX 550, BARRINGTON, ILLINOIS 60010
Phone 312/382-5244 to order

COM-STAR PLUS+ Print Example: ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890

Rational Joystick Interfacing

by Charles Engelsher
Schenectady, New York

A 'Build-it-yourself' project that lets you explore Analog/Digital techniques.

Introduction

Sometimes a seemingly mundane project like joystick interfacing can hold a few educational surprises. For the hardware beginner it provides an opportunity to complete a simple and useful item that is at the same time safe for your Apple and easy on the pocketbook.

More than that, joystick interfacing embraces concepts that have widepread application in other areas of computer hardware: concepts like single-bit A/D conversion, the RC time constant, efficient use of built-in Apple monitor routines, the use of resistive transduction as a basis for measuring physical quantities and proper software scaling of parameters for screen display.

Obviously, this article is more than just a description of a simple project. It provides a vehicle for introducing important hardware ideas to the novice as well.

Analog to Digital Conversion

In the real world, physical quantities vary continuously. A quantity such as temperature or position can literally assume an infinite number of values, even over a narrow range. Digital quantities, on the other hand, vary in a discrete fashion. Simply put, A/D conversion involves translating a continuous physical quantity to a digital format — into zeros and ones.

In all versions of A/D conversion the physical quantity is first converted to an electrical quantity. For instance, a thermistor translates temperature into a resistance, while thermocouple would convert temperature into a voltage level. Such transducers are really doing nothing more than providing an electrical analog of the quantity being measured. This analog signal could be fed to an amplifier, and then to some display device such as a meter.

A/D conversion takes this signal one step further by translating it into digital form. One way of doing this is through multibit conversion, as illustrated in Fig. 1A. You need only supply the right analog signal in the proper range of voltage (or current), and the A/D integrated circuit will output an 8-bit data word which can then be read off the computer data bus. The device shown has an "8-bit resolution." It provides 256 discrete values — 00000000 through 11111111 in binary. This scheme would be quite adequate for temperture measurement between the freezing and boiling points of water. Resolution would be 1 degree Fahrenheit or 1/2 degree Celsius, with a short range on either side of these points.

Now suppose you have only a single output line for the digital data. As you can see from Fig. 1B, the output of such a "single-bit A/D converter" is a simple square wave. This is the situation you're presented with on the Apple game port analog (paddle) inputs. Can you make this singlebit — which can be either HIGH/1 or LOW0 — represent a whole range of values?

Time Constants and Oscillators

Enter the time constant. The solution to the problem of single-bit A/D conversion is to make the length or duration of this square wave vary in proportion to the analog signal. The principle involves varying the resistive-capacitative or RC time constant.

You're probably familiar with mechanical time constants from every day experience. Time constants are best described in terms of exponential rises and falls in some physical property of a system, be it velocity, volume or whatever. Often, the time constant can be used to cause periodic or oscillatory motion in the system. Swinging pendula, flushing toilets and

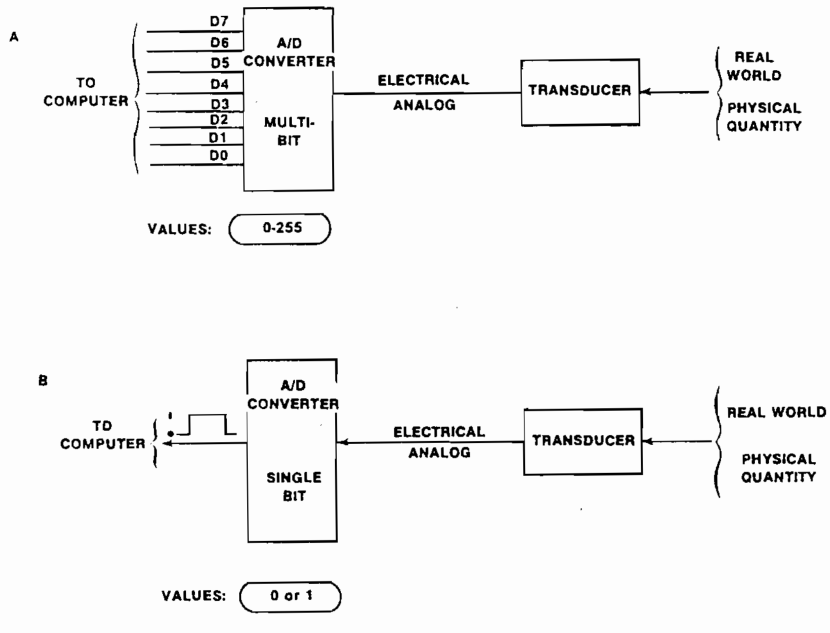


Figure 1

plucked strings all have intrinsic time constants that determine how fast they swing, how quickly they fill and at what pitch they vibrate. The RC time constant, the basis of most single-bit A/D conversion schemes, has the same implications for electrical systems.

The RC time constant determines how quickly a capacitor "fills" with electrons when a voltage is applied. Fig. 2A shows what happens when you apply a voltage to a resistor and capacitor in series. The time constant is equal to the resistance in ohms times capacitance in farads. For the circuit shown, the time constant (TC) is 1 second (1 ohm x 1 farad). The charging curve is shown to the right of the circuit. After two TC's it will be about 86 percent charged. After three TC's the capacitor will be charged up to 95 percent of the applied source voltage (Vs), about .95 volts.

Fig. 2B shows the case for discharge of the capacitor after it has been charged to the applied voltage Vs of 1 volt. In this case it will be about 63 percent discharged after one TC, 86 percent discharged after two and about 95 percent discharged after 3TC's. Notice that the percentage figures are

the same for charging and discharging; only the "direction" is different.

The formulae below the two curves describe the voltage rise and fall in the capacitor in terms of the natural logarithm, e, which equals 2.718. Its reciprocal is .37. If you measure charge or discharge in absolute time T, then the term e-T/RC is used. If you measure time with respect to the number of RC time constants that have passed (N), this term becomes (.37-N).

The key point of all this is that the time constant is a basic property of RC circuits. *If you change either the capacitance or the resistance, you change the time constant.* (At this point we're only a few short steps away from practical single-bit A/D conversion on the Apple, so stay tuned a bit longer.)

Rather than belabor the physics of the situation, consider what happens if you could attach an RC circuit to an active electronic device, one which could provide a periodic charging and discharging current to the capacitor. By connecting a resistor and capacitor to one type of integrated circuit you can produce oscillations.

For the device shown in Fig. 3A, the output will be a train of square waves. The period between successive square waves is indicated by "T" in the figure. The reciprocal of this period is the frequency of the signal. This frequency can be varied by changing R or c. In this circuit a variable resistor or potentiometer (technically called a rheostat) is illustrated, as this is the easiest way to change the frequency of the square wave signal.

The oscillator is not the ideal way to achieve single-bit A/D conversion, however. One reason is that the output frequency of such oscillators (the 555 timer being one example) does not change in direct proportion with changes in the resistance. Another reason is that the software necessary to measure frequency is more complex and takes longer to execute than the preferred method: the ONE-SHOT.

One-Shots

A simplified circuit for the preferred method of single-bit A/D conversion is shown in Fig. 3B. One-shots produce a single square wave pulse when set off by a brief trigger pulse and come in integrated circuit (IC) form. For many commonly available one-shot IC's, *the duration of the square wave output is exactly equal to the time constant*, that is, the product R x C. Not only that, but the variation of the duration of pulse width varies linearly with (in direct proportion to) the variable resistor.

Single-bit A/D conversion using one-shots entails the same steps as that for multi-bit conversion:

1. transduction of a physical quantity into an electrical analog — current to charge the capacitor in this case, followed by
2. production of a digital output — a square wave whose pulse width is proportional to the RC time constant (PW = RC).

With the capacitance value held constant, the pulse width will be proportional to the resistance.

This is exactly the scheme for A/D conversion used on the Apple's game port analog, or paddle, inputs. In the Apple there are actually four one-shots on one integrated circuit, the "quad

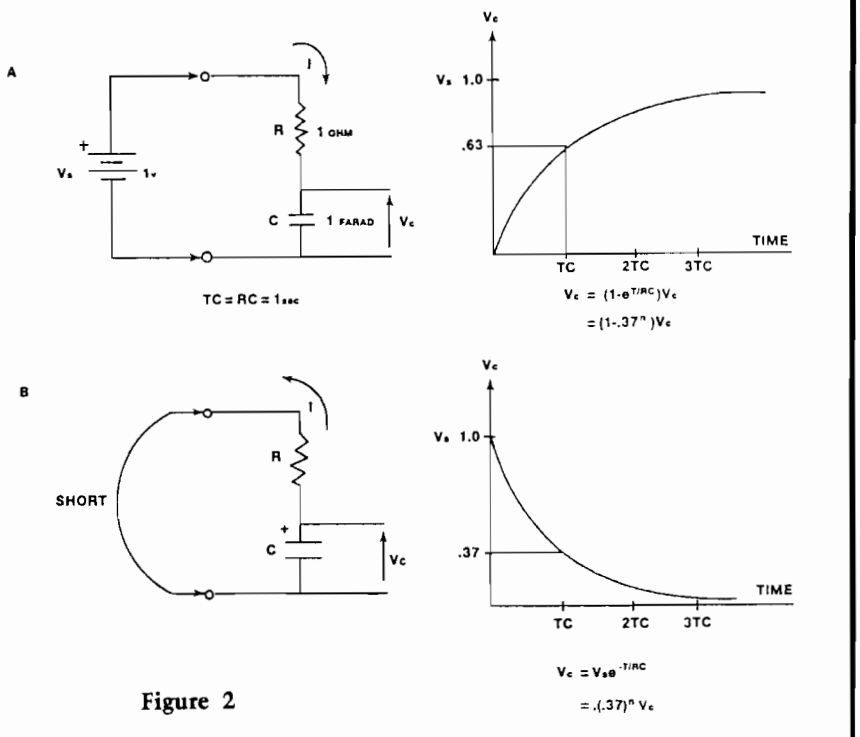


Figure 2

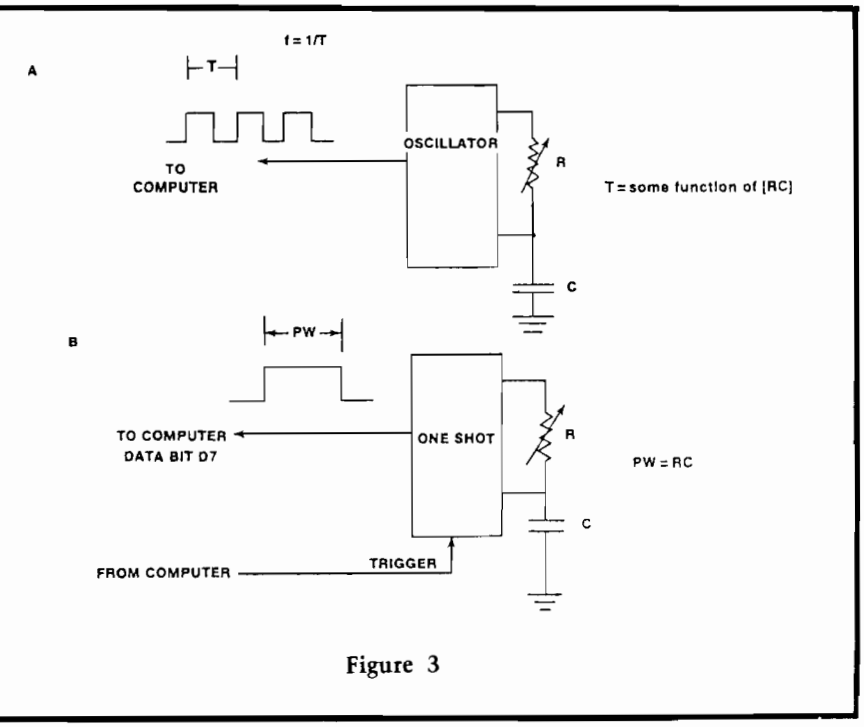


Figure 3

558 timer" as it is called. Each one-shot on this IC is connected to its own pin on the game socket, to which in turn the user plugs in a variable resistor (paddle). Two paddle inputs can be paired, with one for screen x-axis and one for y-axis. This allows for x,y display of a shape on the screen through a joystick or similar device.

The only missing ingredients for our joystick project are the details of the joystick circuitry and software routines used to measure the pulse width and display results to the screen. These two elements are interrelated and will be covered together.

A Practical Joystick Circuit for Apple

Fig. 4 summarizes the main elements in any paddle input A/D conversion scheme: the given software, calculations for matching time constants and the basic circuit.

Let's look at the software first. Apple's PREAD (paddle read) is a built-in monitor subroutine which measures the pulse width of a square wave. PREAD is located at $FB1E (64286 decimal) in the monitor. It must be entered with the paddle number (0, 1, 2 or 3) in the X-register. This is done automatically with the PDL command from BASIC, but must be done "manually" with a LDX command if you are programming in assembler.

The first step in PREAD is to trigger the one-shot so that the square wave output is initiated. This is done by accessing location $C070 (STA would work just well as LDA). The Y-register is used as a counter and is set to zero. The two do-nothing commands (No OPeration) are used to fine tune the counting for the first count.

The body of PREAD is the loop beginning at PREAD2. With the paddle number in the X-register, PREAD2 checks the status of the most significant data bit (D7). If D7 is zero (positive number in binary notation), then the square wave must have returned to zero and you exit PREAD. If D7 is binary one, then the Y counter is incremented and the count continues.

However, if Y is incremented past $FF (255) it becomes zero; this constitutes an overflow and you again exit the routine (after decrementing Y back to 255).

The value of the paddle returned to the calling routine by PREAD will be in the Y-register. The calling routine (e.g., in BASIC or assembler) will then process this value for display or other purposes.

PREAD takes 12 microseconds, or about 3.06 miliseconds for a full count of 255. Compare this with the maximum hardware RC time constant on the paddle. With the standard 150K paddle potentiometer provided in an Apple paddle and the .022 uf internal capacitor on the main board circuitry, the maximum TC is about 3.3 millisec, roughly 8 percent more than the maximum PREAD loop time. This excess is a safety factor; the paddle pot. could fall a bit short in its full scale resistance and still return a full count of 255.

All you have to do is get a surplus 150K joystick and you're in business, right? Not quite. When you go to purchase a "bare" joystick, you'll have one heck of a time finding one with 150K pots. (Unless you want to spring $50 or $60 for a commercial joystick.) Surplus and mail order 100K joysticks are readily available for three to five dollars, however.

Obviously, with such a low resistance, some capacitance must be added to each pot in the joystick in order to bring the maximum RC time constant up to an acceptable level. A value of 3.3 milliseconds for TCmax is used, as this duplicates the safety margin of Apple joysticks. The circuit for one paddle input is given in at the bottom of Fig. 4.

By adding a capacitor, C2, in parallel with the main board capacitor, C1, the TC value of 3.3 msec can be realized. The capacitances are simply additive in parallel configuration. (The small current limiting 100 ohm resistor can be ignored.) The calculation for this added capacitance is given in the figure. For a 100K pot C2 will be .011 uf. A value of .01 uf is a good starting approximation; any additional capacitance can be added in parallel to this if needed.

```
PREAD     LDA   $C070     ;Trigger paddles
          LDY   #$00      ;initialize counter (Y-register)
          NOP             ;compensate for 1st count
          NOP
PREAD2    LDA   $C064,X   ;paddle # in X-register
          BPL   DONE      ;exit if high bit (D7) is zero
          INY             ;otherwise incr. counter
          BNE   PREAD2    ;continue counting          12 msec per cycle
          DEY             ;unless counter is full
DONE      RTS             ;exit, with value (0-255) in Y.

      MAX PREAD LOOP TIME  = 255 CYCLES x 12 microsec/cycle
                           = 3060 microsec or 3.06 millisec

      HARDWARE TIME CONSTANT (as is w. 150K pot) = .022 uf x 150K
                                                 = 3300 microsec

      FOR VALUES LESS THAN 150K USE AN EXTRA CAPACITOR  C2
      IN PARALLEL WITH INTERNAL CAPACITOR  C1 :

            TC  = (C1 + C2) x Rp
         or C2  = TC/Rp - C1)

      For a 100K pot this becomes:

            C2  = 3300 uf/100K - .022 uf
                = .011 uf
```

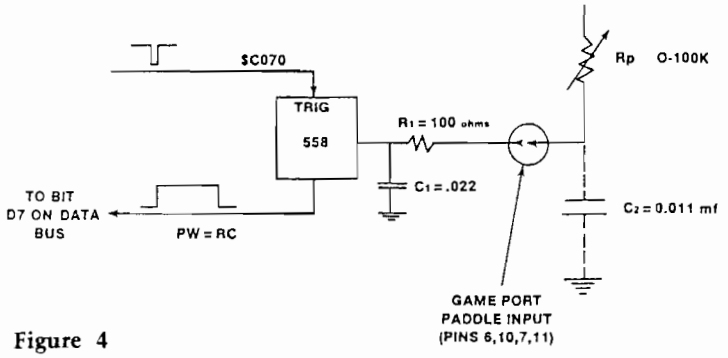


Figure 4

1 - Solderless Breadboard Strip - Radio Shack (Experimenter 300) RS #276-174
1 - RIBBON CABLE JUMPER 16 pin -- 3 ft. double ended jumper
2 - PUSHBUTTONS -- momentary contact, normally open RS #275-1547
1 - 100K JOYSTICK -- Radio Shack #271-1705
2 - 330 ohm resistors -- 220 to 1000 ohm OK
DISK CAPACITORS -- asst of .001 to .1 microfarad

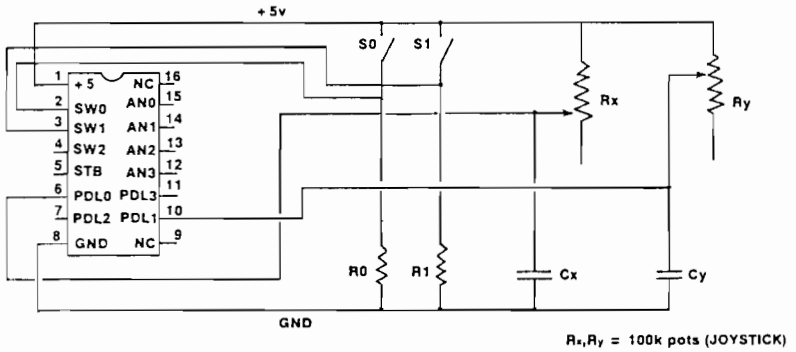


Figure 5

The whole circuit and parts list is given in Fig. 5. The actual cost of the parts used will probably be about $10 to $15 dollars. It's best to use a solderless breadboard strip (Experimenter 300 available from Radio Shack) to breadboard the circuit before wiring it up in a permanent configuration. Plug one end of the dual ended DIP (dual-in-line package) jumper into the game socket on Apple's main board and park the other end into the breadboard. You can omit the pushbutton switches for the initial circuit if you wish.

Arbitrarily, paddle 0 has been assigned as the X-axis paddle and paddle 1 as the Y paddle. These assignments can always be changed later in software if desired.

Software Scaling

Once you've installed the circuit on the solderless breadboard, you must test it out. An easy method entails use of the LORES screen, but keep in mind that with any screen display there is a difference between screen shape and joystick execution. Fig. 6 illustrates this disparity.

Assume for the moment that you're writing the software to display a LORES joystick-moveable screen cursor. With a short routine such as the one in Listing 1, you'll have a clear idea of the values returned as the joystick moves about and within its circular boundary. If you use the maximum values at each edge of X,Y

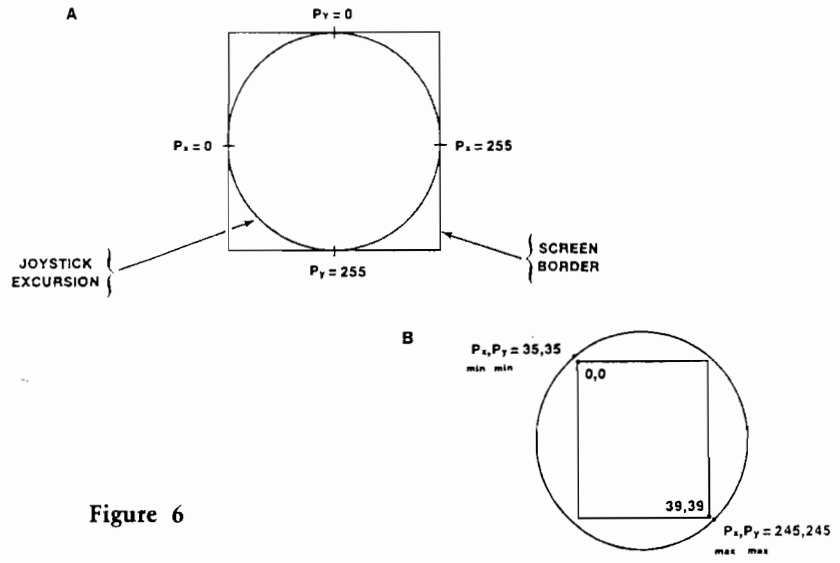


Figure 6

Listing 1

```
96    REM  -------------------
97    REM   PADDLE VALUES
98    REM  -------------------
99    REM
100   HOME
110   PX = PDL (0):PY = PDL (1)
120   PRINT PX,PY
130   GOTO 110
```

paddle excursion (0 and 255), then the cursor will never reach the corners of the screen. The reason for this is given in Fig. 6A.

Instead, you must be sure to scale the values returned from the joystick so that the joystick's excursion includes the corners of the screen. As an example, let's say that the potentiometer X,Y values returned from the upper left hand corner (Px,min and Py,min) are 35 and 35, respectively. Similarly, assume that the maximum paddle values at the lower right hand corner are 245 and 245 (Px,max and Py,max). This is shown in Fig. 6B. Software must convert these values to the graphics screen ranges.

Note: Since you are breadboarding the circuit, you should freely change Cx and Cy and test the Px and Py values returned from Listing 1. Begin with Cx and Cy values calculated earlier. If the .01 uf values are too high, begin with .005 uf and add capacitance in .001 uf increments. This will allow you to achieve the 240 to 250 range for Px,max and Py,max. Once you've empirically optimized Cx and Cy values, you can proceed.

In this example we'll develop a short equation to convert paddle values to mixed LORES coordinates. The equation for the X axis for LORES is given in Fig. 7. The LORES X value will fall in the range of 0 to 39. Sx is the scale factor, Px the current value returned from paddle-X, and Px,min is the value returned from the upper left hand corner of the screen.

The equation for Sx, which depends on Px,min and Px,max, is also given in Fig. 7. Allow a little "dead space" on either side of the extreme points of upper left to lower right excursion; this improves joystick action, as you don't have to jam it at the limit of travel to get the max and min values. Dead space is illustrated in Fig. 6B by the shaded area. Letting Px,max = 240 and Px,min = 40, Sx becomes .196 (raised up one-thousandth.)

$$X = S_x(P_x - P_{x,min})$$

$$S_x = \frac{X_{MAX}}{P_{x,max} - P_{x,min}} = \frac{39}{240-40} = .196$$

PROGRAM STATEMENT: X = INT(.196 * (P_x - 40))

SIMILARLY FOR Y: Y = INT(.196 *(P_Y -40))

Figure 7

In BASIC format, the equation becomes "X=INT(.196*(Px-40))" for the X-axis. A similar calculation and equation is involved for the Y-coordinates (0 to 39 also in mixed mode).

The routine in Listing 2 will display a LORES "boxel" on the screen using the scaling equations and print out the paddle values and X,Y coordinates.

Once you're satisfied with joystick operation, you can hook up the circuit in Fig. 5 as a permanent installation. Use a 16-pin IC socket mounted on a piece of perfboard cut to serve as a lid for a plastic project box. Radio Shack has these materials. Mount the socket at one end and the joystick behind it. The pushbuttons should be mounted on either side of the box underneath the socket. You need only to insert the DIP jumper plug into this homemade joystick and you're in business.

Listing 2

```
96    REM ----------------------
97    REM  LORES JOYSTICK DEMO
98    REM ----------------------
99    REM
100   HOME : GR
110   PX = PDL (0):PY = PDL (1)
120   X = INT (.196 * (PX - 40)):
      Y = INT (.196 * (PY - 40))
130   IF X < 0 THEN X = 0
140   IF X > 39 THEN X = 39
150   IF Y < 0 THEN Y = 0
160   IF Y > 39 THEN Y = 39
170   COLOR= 0: PLOT XT,YT
180   COLOR= 1: PLOT X,Y
190   XT = X:YT = Y
200   PRINT ''PX/X: '';PX;''/'';
      X,''PY/Y: '';PY;''/'';Y
210   GOTO 110
```

Summary and Suggestions

At this point you know how to construct your own joystick for about 1/4 the price of a commercial unit. If you followed the sequence up to joystick construction, you've learned a bit about the RC time constant, one-shots and time interval measurement from assembly language — that is, about single-bit A/D conversion using a resistive transducer. For the example shown, the conversion was from angular position (a continuous real world quantity) to LORES coordinates (digital quantity). Below are a few project suggestions.

You may want to incorporate the method for scale calculation in a routine that automatically calculates the proper scale factors for X and Y. Sx and Sy could then be customized for any joystick, whether homemade or commercial. Such a feature would be useful in games and much appreciated by the game player. Hint: use pushbuttons or keyboard to signal the upper left and lower right corners of the joystick's excursion.

You may also want to apply the same method presented here to develop the scaling equations and BASIC statements for HiRes display. You'll have no trouble with the Y-coordinates (0-191), but since the X-coordinate can range from 0 to 279, some compromise will be necessary. You have two choices:

1. Use the full scale of the X-axis (0 to 279) and calculate Sx accordingly. This method will result in "dropout" or non-plotting of a few points along the X-axis, but it does give a full range of X motion when this is needed.
2. Omit plotting the edges of the X-axis — say 40 points on either side. This leaves 200 points or so in the central portion of the X-axis and prevents full X-axis excursion, which may or may not be a drawback in certain applications. It does, however, provide a point-for-point correspondence between joystick position and screen positioning of a cursor or other shape, a necessary feature in some instances.

An entirely different set of applications for single-bit A/D conversion is the use of transducers other than joysticks. You might want to use a cheap resistive photocell (e.g., Radio Shack 276-116) as a light transducer. Check its resistance in the range of light intensity you want to measure with an ohmeter, and calculate the value of added capacitance if needed. Thermistors, which change resistance in response to temperature changes, provide another possibility for experiment. Brands such as Fenwall are available through over-the-counter and mail order parts distributors.

Those of you who are comfortable with assembly language might want to modify the PREAD routine in one or more of the following ways:

Write a PREAD routine that will use double precision counting, that is, one which increments a 16-bit counter consisting of two memory locations. Ideally, these should be in page zero to increase execution speed. Values returned will be in the range of 0 to 65535. This is a dramatic increase in range of measurement. Naturally, speed will be proportionally reduced so that the sampling rate will be little better than once per second.

Have your modified PREAD store a memory page (256 bytes) or more of data automatically, under user control. This maximizes the speed of PREAD, as all the data is dumped into memory in quick succession. The data can then be analyzed and displayed later on. For proper display, you should make sure that the sampling intervals between each data element are equal.

Finally, for joystick applications, modify PREAD to read the X and Y resistors in sequence. Naturally, two page-zero memory locations are the best choice for the X and Y counters. The advantages of such a dedicated PREAD routine are two: you avoid the problem of inaccuracy when you trigger two paddle inputs in succession, and you significantly increase joystick reading speed.

These suggestions just scratch the surface. Some applications will require a little electronics background. A little study on the use of operational amplifier IC's (which boost sensitivity of certain transducers) will serve you in good stead if you pursue this subject.

Want to become an Apple expert?

Join *the* club.

A.P.P.L.E.

Apple PugetSound Program Library Exchange

The Apple PugetSound Program Library Exchange is the world's first, oldest, and largest Apple computer user group. Our membership is comprised of Apple enthusiasts throughout the world, and we provide support for all levels of technical ability, from beginner to seasoned program author.

A membership in A.P.P.L.E. will bring the Apple owner 7 day per week hotline privileges for technical assistance **when you need it,** plus the international magazine Call—A.P.P.L.E., and incredible discounts on our fully supported, low priced, world famous software products, and hardware.

A.P.P.L.E. is a member owned, non-profit service organization. Write today for a free copy of our magazine and club information, or join by filling out the enrollment coupon

VALUESOFT

A product whose time has come.

Fully supported, guaranteed, high quality software at low prices.

VALUESOFT INCLUDES:

Uncommon at

$12.50 per disk.

program contains its own document on disk.

- 2 word processing programs
- exciting games
- useful utilities
- high quality graphics
- finance and education

The quality VALUESOFT line of software is available through the Apple PugetSound Program Library Exchange.

NEW *IIc* OWNER?

programs for the *IIc* now in stock

Graphics
Word Processing
Utilities
Database

Join Now and Receive 10 FREE (5-1/4")

Apple][ ][+ //e //c /// Lisa and Macintosh are all registered trademarks of Apple Computer Inc.

A.P.P.L.E.
pioneering Apple computing

since 1978.

Mail to
A.P.P.L.E.
21246 · 68th Ave. S.
Kent, WA 98032
(206) 872-2245
or call our toll-free number
1-800-426-3667
(24 Hrs. Orders Only)

☐ MEMBERSHIP $26 one-time application fee + $25 first year dues. $51

☐ FREE INFO + Call—A.P.P.L.E.
Please send free information

Name _____

Address _____

City _____

State _____ Zip _____

Phone # _____

M/C VISA # _____

Exp. Date _____

Additional foreign postage required for membership outside the U.S.

68000 p-System BASIC

by Paul Lamar
Rendondo Beach, California

& Charmaine Lindsay
Willowdale, Ontario, Canada

There is nothing more boring than a standardized computer unless it is a standardized high level language. Computer hardware design, operating system design and high level language design is almost an art form. Just the right combination of characters and lines on the screen, the right feel to the keyboard, the proper ergonomics, etc. You can always use more speed, RAM and disk storage, of course. There are those who would standardize both hardware (IBM PC ??) and software. This I think would be a mistake. Computer design, operating systems and high level languages are living and growing things just as are common languages (English for example). With 16 megabytes of memory addressing on the 68000 microprocessor and 100 megabyte, five inch winchester hard disks available, we have a long way to go in micros, micro operating systems and in high level computer languages. I don't think we should stop here.

BASIC is the easiest to use, most versatile High Level Language (HLL) so far. It is easy to learn and fun to write short programs in. This article is about a better BASIC.

The p-System has a very nice compiler BASIC written by Softech Microsystems. This BASIC will run on any machine that runs the p-System. That includes, but is not limited to, 68000 SAGE, Pinnacle, Micro Craft Dimension, Hewlett Packard 9816, Corvus Concept, Analytical Engines Saybrook and last, but not least, the 8088 IBM PC and all its clones that have implemented the p-System. Needless to say, it runs about four times faster on a 8 megahertz 68000 such as the SAGE than it does on an IBM PC. The p-System BASIC has all the usual basic commands and constructs such as IF THEN ELSE, FOR TO STEP NEXT, ON GOTO and ON GOSUB, PRINT USING etc. In addition, it has some unusual extensions.

Units

The UNIT statement is very unusual and can be used to write BASIC programs that can be called from another BASIC program, or a FORTRAN or Pascal program. This is done by storing the BASIC program UNIT on disk or in RAM disk within a "LIBRARY" of other UNITS. UNITS are a concept originally borrowed from assembly language by Pascal. The author of Pascal merely changed the name from "Called Module" to "UNIT". This is quite typical of Pascal and other high level languages. UNITS are a form of modular programing because they can be separately compiled and debugged in the same way as Called Modules can be separately assembled and debugged in assembly language.

Believe it or not, you can call a Pascal subroutine (called a PROCEDURE in Pascal, subroutines being dirty words in Pascal) from within this wonderful BASIC. There are quite a few Pascal PROCEDURES available to run on the p-System that you may want to use in your BASIC programs. The way you do this is as follows:

```
REM  These commands are imbedded in the BASIC program.

LIBRARY "UNIT2.LIBRARY"
(This is the name of a file on disk or in RAM disk that holds a selection of "UNITS". These "UNITS" are separate, short PASCAL programs that have PROCEDURES (subroutines) imbedded within them, just as a BASIC program has subroutines imbedded within).

USES FANCONTROL
(FANCONTROL is the name of a UNIT that controls the fan. What else ?)

CALL FANON
(Finally we get around to calling the actual subroutine (I'm sorry ..... PROCEDURE ... shame on you, Paul) that does the trick.)
```

We only have to use "LIBRARY" and "USES" once, at the beginning of our BASIC program. From then on we can "CALL" FANON as many times as we like.

Line Numbers

A very nice real feature of this BASIC is the fact that it does not need line numbers. Line numbers are only needed for GOTOs, Subroutines and IF THEN ELSE. This feature allows rapid programing because you can use the editor to replicate blocks of code. To illustrate, let's take a block of code such as Figure 1. This block of code is a general purpose input routine with error checking. If we use the program editor's copy feature to repeat this block of code over and over again we can, with minor modifications to the block, use this same block to enter many different variables into our program. After we replicate the block, we put the editor in the exchange mode and run the cursor down changing one character in each different variable name (see Figure 2.)

Another advantage of the scarcity of line numbers is the possibility of rearranging the code after you have finished programing. Rearranging the code also makes it much more understandable and easier to follow. Rearranging the code makes the program neat and progress in a logical manner, one of the claimed advantages of Pascal. No need to learn structured programing. Forget all that "top-down" nonsense. No need to structure your program, just blurt out your ideas. Get that tricky subroutine, or algorithm you having been thinking about, off your chest. You can clean it up later with this editor. This is the way to have fun, enjoy life and experience real freedom in programming.

You move code in the editor by first deleting the block of code that you would like to move. Sounds contradictory but don't worry it works. You will get used to it. What really happens is all code that is deleted goes into a Copy Buffer in RAM. It stays there until you delete the next block of code which will then overwrite the Copy Buffer. To get the last block of code, that you deleted back out of the Copy Buffer, you put the cursor where you would like to have that block of code. You then press "c" for copy. The editor prompts "Buffer or File Name" Press b for the Copy Buffer and the code magically reappears where you want it.

The same sequence of commands works for replicating blocks of code as well as for moving them. We are digressing into the operation of the p-System editor which is a whole other story by itself. Back to p-System BASIC.

Variables

Another nice feature of this BASIC is that variables are significant up to 8 characters. This makes the code largely self commenting. It also makes it quite easy to come up with new meaningful names and lessens the need for local variable names. The worst of the early BASICs only allowed one letter followed by one number (A1, G9, ... etc.). Later BASICs, such as Applesoft, would allow as many letters as you like but would only consider the first two as significant. BUTTOCKS and BUSTY are the same variable names in Applesoft. (That's the one remaining trouble with computers, they lack sensuality.)

Display and Print

DISPLAY of course, displays it on the CRT and PRINT actually prints the results on a real printer. The PRINT command in BASIC at first confused and later amused me. When I first started to program in BASIC on a CRT ten years ago, I always wondered why it was called "PRINT". It did not seem right to call it PRINT when that was done on a "printer" not on a CRT. The use of the word PRINT in BASIC stemmed from the early days, twenty years ago of the language when all they had was a teletype printer as input/output for their computers running BASIC. DISPLAY makes more sense if you are going to display it on a CRT.

Image

The IMAGE statement appears to me an another unusual feature. I am not what anybody would call an expert on BASIC languages. I am familiar with Microsoft basic as implemented on the Apple, Commodore and CP/M. IMAGE to my knowledge does not appear in Microsoft basic. The IMAGE statement is referred to by line number within the USING clause of a DISPLAY or PRINT statement. For examples:

Example 1:

A = 88.888 (note the lack of a line number and the LET statement)
10 IMAGE *###.##* (line number required here)
PRINT USING 10:A (again no line number)

The printer prints "88.89" with the .888 rounded to .89.


```
Basic Compiler  IV.Ø b5-4     SYSTEM.WRK.TEXT
```

Figure 1

```
  1   2    1:Ø     Ø  rem fig. 1
  2   2    1:Ø    12
  3   2    1:Ø    12  input "1st   variable ? ":variable1
  4   2    1:Ø    43
  5   2    1:Ø    43
  6   2    1:Ø    43
  7   2    1:Ø    43  rem fig. 2.
  8   2    1:Ø    43
  9   2    1:Ø    43
 1Ø   2    1:Ø    43  input "1st   variable ? ":variable1
 11   2    1:Ø    74  input "2nd   variable ? ":variable2
 12   2    1:Ø   1Ø5  input "3rd   variable ? ":variable3
 13   2    1:Ø   136  input "4th   variable ? ":variable4
 14   2    1:Ø   167  input "5th   variable ? ":variable5
 15   2    1:Ø   198  input "6th   variable ? ":variable6
 16   2    1:Ø   229  input "7th   variable ? ":variable7
 17   2    1:Ø   26Ø  input "8th   variable ? ":variable8
 18   2    1:Ø   291
 19   2    1:Ø   291
 2Ø   2    1:Ø   291  end
```

Figure 2

Example 2:

S$ = "The subtotal is"
DISPLAY USING 10:S$,A

"The subtotal is 88.89", is displayed on the CRT.

Example 3:

20 IMAGE $$###.## {The double dollar sign indicates a floating $}
PRINT USING 20:S$,A

The printer prints "The subtotal is $88.89".

You don't have to use IMAGE. PRINT and DISPLAY USING works in all the usual ways just like Microsoft BASIC.

Disk File Handling

There are the usual OPEN and CLOSE statements for RELATIVE and SEQUENTIAL disk files. RELATIVE files allow sequential and random access at the expense of extra verbiage. SEQUENTIAL files are of fixed or variable length. There is a RESTORE statement that is used to reposition the internal file pointer to the first record within a SEQUENTIAL disk file. The RESTORE statement is used to reposition the pointer to a specific record within a RELATIVE file.

The ASSIGN statement is used to set up an array on disk just as most BASICs set up ordinary arrays in RAM. This frees up some RAM for use by the program. It also results in a permanent record of the array in case of power failure. Having an array on disk is slower, of course, than having it in RAM. The ASSIGN statement seems a little redundunt with the 68000 as the 68000 can directly address 16 megabytes of RAM. Perhaps this feature would become more useful if the maximum size of the arrays was 10 mega elements rather than only 32K elements. That way a hard disk could be used to real advantage. The RAM disk option can be used with the ASSIGN statement to allow larger programs that run almost as fast as arrays located in ordinary RAM. Unfortunately, considering the large memory addressing capabilities of the 68000, arrays larger than 32K are unallowed in ordinary RAM as well.

Ease and Speed of Programing

When writing BASIC programs that needed compiling on lesser operating systems than the 68000 running the p-System you had to: write the program, save the resulting text file on disk, load the compiler from disk, reload the BASIC text file from disk, compile the code saving the resulting code file on disk, load the compiled code from disk and run the program. This was a long and tedious process usually taking several minutes. On the 68000 p-System you merely press "Q" for quit the editor, "U" for update the work file [a temporary file in RAM] and "R" for run the program. The text file will automatically be compiled, saved on disk and the program will run very quickly. These prompts are always on the screen in case you forget.

If you have a syntax error in your program, the compiler will stop and prompt: Continue ? Quit ? or Edit ? If you press "e" or "E" for edit the BASIC text file will be automatically reloaded from RAM disk in several seconds and the cursor will be placed just beyond the error. This is really nice if you program the way I do, by the trial and error method.

The Documentation

An interesting aspect of this BASIC is that: in some cases the description of the constructs and statements are written from the point of view of the Pascal programer. It is as if the authors of this BASIC finally realized the limitations of Pascal and set out to write a BASIC that incorporated the better aspects of UCSD Pascal. For example, when discussing subroutines the authors immediately lapse into a discussion of "procedure blocks". For some reason they could not quite bring themselves to just call it a subroutine. By the way, there are subroutines in assembly language and they work exactly the same way as they work in BASIC.

On the whole the documentation is quite good with lots of examples. However it is not written for beginners and some knowledge of Pascal would be helpful for those few constructs that are similar to Pascal.

```
Basic Compiler  IV.Ø b5-4     SYSTEM.WRK.TEXT


    1   2    1:Ø     Ø  {$N+}
    2   2    1:Ø    22  for x=1 to 9
    3   2    1:Ø    57  display "paul"
    4   2    1:Ø    74  next x
    5   2    1:Ø    76  end
```

Figure 3

Figure 4

```
================================================================
Final MC68ØØØ IV.Ø2 [a.1] Code for              RADIX  1Ø
 segment PROGRAM  procedure   1  MP          .EQU    AØ
 segment word offset    17       BASE        .EQU    A1
Source Object                   SEG         .EQU    A2
P-Code N-Code                   PME         .EQU    A3
(Dec. Offsets)                  DATA        .EQU    A6
================================================================

         Ø                                  .WORD   2Ø4,Ø
         Ø
    4:   Ø 86Ø1              ;p-code    LAO     1
         2 ØØ                ;p-code    SLDC    Ø
         3 811ØØ             ;p-code    LDCI    2Ø
         6 ØØ                ;p-code    SLDC    Ø
         7 7Ø15              ;p-code    SCXG    KERNAL   ,21
         9 ØØ                ;p-code    SLDC    Ø
        1Ø A5Ø1              ;p-code    SRO     1
   19:  12 A8                ;p-code    NATIVE
        13 A8                ;p-code    NATIVE
        14 49E9 ØØØC                    LEA     12(BASE),A4
        18 99CE                         SUBA.L  DATA,A4
        2Ø 3ØØC                         MOVE.W  A4,DØ
        22 3340 ØØ1C                    MOVE.W  DØ,28(BASE)
```

continued

The Native Code Generator

This feature of the UCSD p-System is the one feature that I am most excited about. The Native Code Generator, generates a partial assembly language text file from a BASIC program listing. See Figures 3 and 4. Unfortunately, at this time the file is part assembly language and part p-code text file, (the BASIC compiler normally generates p-code which is executed by a p-code interpreter). At the present time, June 1984, there is a bug in the Native Code Generator that does not handle backward GOTOs in BASIC properly.

The original purpose of the Native Code Generator was to bypass the p-code interpreter and thereby speed up the execution of all high level languages running under the p-System. This is a worthwhile and highly desireable goal for the Native Code Generator. The use I have in mind is quite different however.

In way of explanation the high level language "C" also generates an assembly language text file which must then be run through an assembler to generate an executable code file. This intermediate assemble language text file form gives the programer the opportunity to exercise detailed control over the speed of execution of all parts of his program. This is the reason that "C" is the preferred high level language for writing operating systems that control time critical hardware such as disk drives and printers. My own opinion is that time critical operating systems are best written in assembly language.

As a general purpose high level language "C" is a little too cryptic for my taste. I prefer BASIC. In Softech p-System BASIC however, running the Native Code Generator generated assembly language text file through an assembler is not possible at the present time. If the Native Code Generator was expanded to convert the entire BASIC text file to an assembly language text file it would then be possible to run that assembly language text file through an assembler and generate a executable machine language code file. The same as in "C". This would result in unprecedented flexability for this BASIC. A programer could optimize the resulting assembly language text file for speed or change it for detailed hardware control. This would give the best of both worlds, the speed, flexability

Figure 4 *continued*

```
26:    26 3F29 ØØ1C              MOVE.W  28(BASE),-(SP)
       3Ø 3F3C ØØØ1              MOVE.W  #1,-(SP)
       34 4EAB ØØØ8              JSR     8(PME)
       38 CC           ;p-code   FLT
       39 F4           ;p-code   STRL
31:    4Ø A8           ;p-code   NATIVE
       41 A8           ;p-code   NATIVE
       42 49E9 ØØ14              LEA     2Ø(BASE),A4
       46 99CE                   SUBA.L  DATA,A4
       48 3FØC                   MOVE.W  A4,-(SP)
       5Ø 3F3C ØØØ9              MOVE.W  #9,-(SP)
       54 4EAB ØØØ8              JSR     8(PME)
       58 CC           ;p-code   FLT
       59 F4           ;p-code   STRL
36:    6Ø A8           ;p-code   NATIVE
       61 A8           ;p-code   NATIVE
       62 6Ø1A                   BRA     L1
38:    64 3F29 ØØ1C    L3:       MOVE.W  28(BASE),-(SP)
       68 3F29 ØØ1C              MOVE.W  28(BASE),-(SP)
       72 4EAB ØØØ8              JSR     8(PME)
       76 F3           ;p-code   LDRL
       77 A8           ;p-code   NATIVE
       78 3F3C ØØØ1              MOVE.W  #1,-(SP)
       82 4EAB ØØØ8              JSR     8(PME)
       86 CC           ;p-code   FLT
       87 CØ           ;p-code   ADR
       88 F4           ;p-code   STRL
       89 A8           ;p-code   NATIVE
47:    9Ø 3F29 ØØ1C    L1:       MOVE.W  28(BASE),-(SP)
       94 4EAB ØØØ8              JSR     8(PME)
       98 F3           ;p-code   LDRL
       99 A8           ;p-code   NATIVE
      1ØØ 49E9 ØØ14              LEA     2Ø(BASE),A4
      1Ø4 99CE                   SUBA.L  DATA,A4
      1Ø6 3FØC                   MOVE.W  A4,-(SP)
      1Ø8 4EAB ØØØ8              JSR     8(PME)
      112 F3           ;p-code   LDRL
      113 CE           ;p-code   LEREAL
      114 A8           ;p-code   NATIVE
      115 A8           ;p-code   NATIVE
      116 3Ø1F                   MOVE.W  (SP)+,DØ
      118 E258                   ROR.W   #1,DØ
      12Ø 642C                   BCC     L2
57:   122 4EAB ØØØ8              JSR     8(PME)
      126 721A         ;p-code   SCXG    BLIB    ,26
59:   128 7225         ;p-code   SCXG    BLIB    ,37
61:   13Ø 7247         ;p-code   SCXG    BLIB    ,71
63:   132 A8           ;p-code   NATIVE
      133 A8           ;p-code   NATIVE
      134 3F3C ØØ18              MOVE.W  #24,-(SP)
      138 3F3C ØØØ8              MOVE.W  #8,-(SP)
      142 4EAB ØØØ8              JSR     8(PME)
      146 9D           ;p-code   LPR
      147 A8           ;p-code   NATIVE
      148 3F3C ØØ1Ø              MOVE.W  #16,-(SP)
      152 4EAB ØØØ8              JSR     8(PME)
      156 7226         ;p-code   SCXG    BLIB    ,38
7Ø:   158 7247         ;p-code   SCXG    BLIB    ,71
72:   16Ø 722A         ;p-code   SCXG    BLIB    ,42
74:   162 A8           ;p-code   NATIVE
      163 A8           ;p-code   NATIVE
      164 6Ø9A                   BRA     L3
76:   166 4EAB ØØØ8    L2:       JSR     8(PME)
78:   17Ø              ;exit code
      17Ø 96ØØ         ;p-code   RPU     Ø
```

and detailed control over the speed of execution of assembly language with the ease of programing in a simple, easy to learn high level language. This BASIC in conjunction with the p-System Native Code Generator is very close to this utopia.

Speed of Execution

Speed of execution is not what BASICs are known for. No one should write a word processing, spreadsheet, data base or spelling checker program in any BASIC, interpreted or compiled. The presently available compilers are just not as efficient as a good assembly language programer at writing code. Nevertheless, the 8 megahertz 68000 p-System compiled BASIC in p-code form is three times faster than Microsoft BASIC running on the IBM PC. Running the p-code through the 68000 Native Code Generator would speed it up by another factor of three except for the floating point routines. These routines are not changed by the Native Code Generator. Fortunately, number crunching is best done by hardware floating point chips.

National Semiconductor has a floating point chip (16081 or 32081) that works quite well with the 68000. Ironically it works faster with the 68000 than with Nationals own 16 bit microprocessor the 16032. This chip is capable of dividing a 64 bit floating point number by another 64 bit floating point number in approximately 30 microseconds with the 68000 running at 10 MHZ and the 16081 running at 5 MHZ. That is the total time required to load and retrieve the operands and store them in main memory. The 16081 requires only 11 microseconds to do the actual divide. The 16081 will shortly be available in a 10 MHZ version. This will not reduce the time by half but by somewhat less than half. The reason is that some finite time, determined by the speed of the 68000, is necessary to load the operands into the floating point chip. Eventually, p-System BASIC, or some other BASIC will be available that will support the 68000/16081 combination.

Converting Microsoft Basic to Softech p-System Basic

Using the following rules, with wise use of the p-System editor commands, you will be able to convert a typical Microsoft program to Softech p-System BASIC in short order. Use them in the order shown to avoid confusion and mistakes. These procedures were provided by Peggy Lakey at SAGE Computer, Reno, Nevada.

You can transfer your Microsoft BASIC text files to the p-System receiving computer using a RS232 serial printer interface cable on the transmitting computer. Connect this cable to a RS232 port on the receiving computer. Any text file that can be printed on your present printer can be transferred. The p-System in the SAGE has a utility program called "TEXTIN". This program will convert an ASCII file to the p-System file format. The resulting Microsoft BASIC text file in the p-System computer can then be loaded into the editor and converted to a p-System BASIC text file using the following rules.

1. Change all occurences of ':' to '::'. Then search for each of these and change back those in strings and quotes, which should not have been altered.
2. Put an 'END' statement at the end of the program, if not already present. All other 'END' statements should be changed to 'STOP'.
3. A statement such as 'DEFINT I-N' should be re-written to read 'INTEGER I, J, K, L, M, N'.
4. Any statement 'IF ... GOTO ..' should be changed to 'IF ... THEN..'.
5. Strings and literal quotes may not be continued from line to line. They must be presented on one line, or displayed in segments, or concatenated.
6. 'TAB (X)' must be followed by ';'.
7. When 'INPUT' is used with an imbedded string as prompt, use ':' before the variable label, not ';'.
8. Change all occurences of 'PRINT' to 'DISPLAY' unless you specifically want an output to go to the printer.
9. Any time a single statement must be continued from one line to another, use the comment delimiters '(* < CR> ' to hide the carriage return from the compiler.

Example:

```
1000 IF x < y then (*
     *) GOSUB 300 : : DISPLAY x
```

10. When multiple statements follow an 'IF' statement on a line, those statements will not be skipped if the 'IF' statement proves false. The following line of Microsoft:

```
20 X = Y : IF X <> 0 THEN X = Z :
   GOTO 5
30 ...
```

should be re-written as follows:

```
20 X = Y :: IF X = 0 THEN 30
   X = Z :: GOTO 5
30 ...
```

11. Change any command that is intended to clear the screen to 'DISPLAY ERASE ALL: '
12. Any command intended to display at a given place on the screen, such as 'PRINT@ 342', or 'HTAB 5 : VTAB 6 :PRINT' should be changed to 'DISPLAY AT (line, column):'.
13. When concatenating strings, use '&', not '+'.
14. Change 'MID$' to 'SEG$'.
15. Change 'LEFTS (X$,L)' to 'SEG$ (X$,1,L)'.
16. Change 'RIGHTS$ (X$,L)' to 'SEG$ (X$,(LEN(X$)-(L-1)),L)'.
17. Change 'GET' to 'INKEY$ (0)'. (NOTE: these statements do not always act exactly alike. Check the definitions in BASIC you are converting from.)
18. Subscripted and non-subscripted variables of the same name are not allowed in the same program. For example, if the variable 'A(10)' and 'A' are in the same program, change them to 'A(10)', and 'A__NUTHER'.
19. In a DIM statement, a variable name may not be used in the parenthesis for dimension size. Thus, DIM X(B,C) is illegal. DIM X(3,4) would be ok.
20. Change all function statements and calls as follows: 'DEF FN F__name(args)' is changed to 'DEF f__name(args)'. (add a FNEND for multiple line functions, see manual) 'FN f__name' is changed to just 'f__name'. Be sure all function names have unique names, and don't use regular variable names.
21. All strings in data statements must have "quotes"d around them.
22. In a DIM statement, a variable name may NOT be used for the array size.

Exec File Utilities

by N. D. Greene
Storrs, Connecticut

A Collection of Eight Useful Exec Utilities for the Apple

Introduction

Here are eight exec file utilities which are useful in writing and examining programs on the Apple II. They may be easily entered and saved with the Textfile Write Edit Read Program (T.W.E.R.P.) described in the September 1984 issue of Micro.

Exec files are text files containing basic commands and/or program line statements. When these files are activated by the EXEC ("execute") command, they mimic direct keyboard entry. If an exec file contains only commands, it is possible to exec it without disturbing any program in memory. For example, one of the utilities described below prints a memory map of the current program without altering it. However, exec files containing program line statements can be used to quickly change the contents of a program. Some examples of this approach are also described below. Further details about exec files and their characteristics are discussed in the disk operating manual.

Description

The exec files described here range from simple command statements to more complex forms. For convenience, they have been divided into three general categories.

Simple Command Routines

Listings 1 and 2 show two simple exec files which may be used to display the contents of high resolution graphics pages one or two without erasing them. This is in contrast to the normal HGR and HGR2 commands which activate page 1 or 2 after erasing their contents. These files should be entered as listed using T.W.E.R.P. or other text file writer. Note that the bracketed 1 with the arrow is used to indicate the first field and should not be entered. The question mark, ?, is the shorthand equivalent of the print command.

Pointer-Based Routines

Certain memory addresses indicate the locations of program elements or variables. These addresses are called "pointers" and may be used as the basis of an exec file. Two examples are shown in Listings 3 and 4. Executing E.BIN (Listing 3) prints the address and length of the last loaded binary file. Listing 4 is a shortened version of a previously published memory map exec file (MICRO 43, Dec. 1981). It finds and displays the program beginning, the program end and other information by peeking at the appropriate pointer addresses. Both of these files may be used without disturbing a program in memory.

Piggyback Routines

Some statements such as INPUT can not be used as direct keyboard commands. These deferred-execution commands must be used within a program. The same restrictions apply to exec files. However, it is possible to use these restricted commands within an exec file without permanently altering a program in memory. This is the piggyback routine. The exec file adds some new lines to the existing program, runs these lines and then deletes them leaving the program in its original form. Typically, high line numbers are chosen to avoid conflict with existing line numbers.

Listings 5-8 show several exec files which use deferred-execution commands via the piggyback method. E.PTR (Listing 5) shows the contents of a pointer address. The initial (low byte) address is input and the program calculates and prints the contents of the two-byte address. If 103 is input, the program start memory will be shown. E.PTR adds three new high-numbered lines, the last one containing a delete command. The file then runs the lines which ask for an input, print the results and then conveniently self-destruct. Actually E.PTR represents a combination of a piggyback and pointer exec file.

E.CTR (Listing 6) makes control characters visible. Control characters, which are normally invisible in catalog and program listings, appear as flashing characters after this file is executed. This is a useful routine to (1) check that all DOS commands in a program contain a control D and (2) find hidden control characters in catalog names. If you accidently insert a control character while saving a program on disk, it will not load unless the invisible character is inserted at the proper point. This is a frustrating experience and a trick used by some commercial programs to prevent listing. E.CTR is a compressed, piggyback exec file version of a conventional program listed on page 151 of reference 2. This file writes a one line program at 63999 (the highest program line permitted in floating point basic) and then runs it. The line deletes itself when the last statement is encountered. Pressing RESET or PR#0 restores normal printouts.

Two "capture" files are shown in Listings 7 and 8. These exec files transfer floating point and integer programs into text files. But why? There are two major uses for these files: 1) to create a library of auto-writing subroutines and 2) to convert integer programs into floating point programs. Both E.CAPA and E.CAPI may be used to create a library of subroutine exec files. The desired subroutine is isolated by deleting all other lines and then captured in a file by executing the appropriate exec routine (E.CAPA for Applesoft floating point programs or E.CAPI for integer programs.) E.CAPA permits a choice of file names; E. CAPI always creates a file names "TEXT.I". Once created, these files may be used to add subroutines to new programs. E.CAPA is a one line, self-destructing, piggyback exec file similar to E.CTR. E.CAPI is a piggyback version of a program listed in page 76 of reference 2. The first line (field) contains an invisible, control D between the quote marks.

E.CAPI may also be used to transform integer into floating point programs. The procedure is as follows. After installing integer via a language or hardware card, load the integer program to be captured. Then exec E.CAPI which puts the program into the exec file, TEXT.I, and saves it on the disk. Next, convert to floating point and

```
1=> POKE49239,Ø:POKE49236,Ø:POKE49234,Ø:POKE49232,Ø
```

Listing 1. E.GR1

```
1=> POKE49239,Ø:POKE29237,Ø:POKE49234,Ø:POKE49232,Ø:END
```

Listing 2. E.GR2

```
1=> HOME:?"ADD=";PEEK(43634)+PEEK(43635)*256
2=> VTAB3 :?"LEN=";PEEK(43616)+PEEK(43617)*256
```

Listing 3. E.BIN

```
1=> HOME:?"HI MEMORY= ";PEEK(115)+PEEK(116)*256
2=> VTAB2:?"STRINGEND= ";PEEK(111)+PEEK(112)*256
3=> VTAB3:?"FREESPACE= ";
  PEEK(111)+PEEK(112)*256-(PEEK(1Ø9)+PEEK(11Ø)*256)
4=> VTAB4:?"ARRAY END= ";PEEK(1Ø9)+PEEK(11Ø)*256
5=> VTAB5:?"ARRAY BEG= ";PEEK(1Ø7)+PEEK(1Ø8)*256
6=> VTAB6:?"LO MEMORY= ";PEEK(1Ø5)+PEEK(1Ø6)*256
7=> VTAB7:?"PROG END = ";PEEK(175)+PEEK(176)*256
8=> VTAB8:?"PROG BEG = ";PEEK(1Ø3)+PEEK(1Ø4)*256
```

Listing 4. E.MEM

```
[1]=> 6ØØØØ HOME:INPUT"A=";A
[2]=> 61ØØØ HOME:?PEEK(A)+PEEK(A+1)*256
[3]=> 62ØØØ DEL 6ØØØØ,62ØØØ
[4]=> RUN6ØØØØ
```

Listing 5. E.PTR

```
1=> 63999 DATA2Ø1,141,24Ø,21,2Ø1,136,24Ø,17,2Ø1,128,144,13,
  2Ø1,16Ø,176,9,72,132,53,56,233,64,76,249,253,76,24Ø,253:
  FORI=768TO768+27:READV:POKEI,V:NEXT:
  POKE54,Ø:POKE55,3:CALL1ØØ2:DEL63999,63999
2=> RUN63999
```

Listing 6. E.CTR

```
1=> 63999 D$=CHR$(13)+CHR$(4):HOME:VTAB12:
  INPUT"CAPTURE FILE NAME==> ";F$:?D$;"ØPEN";F$:?D$;"WRITE";F$:
  POKE33,3Ø:LISTØ,63998:?D$;"CLOSE";F$:TEXT:DEL63999,63999:END
2=> RUN63999:END
```

Listing 7. E.CAPA

```
1=> 32762 D$=""
2=> 32763 PRINT D$;"OPEN TEXT.I"
3=> 32764 PRINT D$;"WRITE TEXT.I"
4=> 32765 POKE33,3Ø:LIST Ø,32761
5=> 32766 PRINT D$;"CLOSE TEXT.I"
6=> 32767 TEXT:END
7=> RUN 32762
8=> DEL 32762,32767
```

Listing 8. E.CAPI

execute TEXT.I, which loads the captured program into memory. At this point, the original integer program is now a floating point program which may be saved in the conventional manner. It is important to remember that there are differences between the commands used by integer and floating point programs. The new, floating point version may not work. If so, it will require appropriate corrections. However, E.CAPI saves the time of reentering and existing integer program. This procedure works because the floating point language does not check for syntax errors until the program is run. (Fortunate for this approach - unfortunate for programmers!) The integer language checks for errors during listing, so it is not possible to use E.CAPA to transform floating point into integer programs.

Applications

Exec files can be a very powerful programming tool. I have a working disk containing E.MEM, E.GR1, E.GR2, E.CTR and E.BIN. Thus, I can examine memory, look at the contents of either graphics page, "see" hidden control characters or find the address and length of any binary file whenever needed and without disturbing the program I have in memory. The names used here are to remind me these are exec files — they could be saved under any other valid name.

I have a second disk containing an extensive library of exec file subroutines created by E.CAPA. These have titles indicating their function and line number range. Some examples are: TITLE CENTER 1000-1005; TIME DELAY 3000-3010; and INPUT 7000-7070. Thus, when writing programs, I auto-write any subroutines I need by executing the appropriate file. After finishing the program, I compress it by using the renumber routine. Of course, it is necessary to keep track of, or use a common set of, variables for subroutines which are added to an existing program.

Exec files are both powerful and dangerous since they mimic keyboard input. This is especially true of piggyback and other exec routines which modify programs. Be careful when using a file the first time. Also be conscious of the changes caused by executing a file. If 176 is input to E.PTR (Listing 5) to find the end of the program in memory, the answer will be incorrect since E.PTR adds three new lines during its execution. In contrast, E.MEM (Listing 4) which does not define any new variables, nor add any program lines, will show the correct value.

There are numerous other possible exec file utilities which could be written. An obvious example is the codepokes program on page 77 of reference 2 which converts machine-language routines into a series of basic poke commands.

References

1. N. D. Greene, *Textfile Write Edit Read Program (T.W.E.R.P.)*, MICRO, No. 75, September 1983, p. 27.
2. *Apple II — The DOS Manual*, Apple Computer, Inc., Cupertino, CA 1981.
3. N. D. Greene, *Applesoft Memory Map Display*, MICRO, No. 43, December 1981, p. 96.

Expanding the Commodore 1541 Disk Drive

by Michael G. Peltier
Wichita, Kansas

A Multi-Part Series on the 1541. Part 1: Expanding the User Commands

Introduction

This is the first part of a three part article discussing ways to expand the Commodore 1541 Disk Drive. Part 1 covers expanding the DOS by explaining the operation of the "**U0**" command and showing how to install the "**UA**" thru "**UP**" commands. Part 2 will explain how to expand the disk drive's RAM, including theory, construction, installation and testing of a 4K expansion RAM. The advantages and disadvantages of expanding the RAM will be discussed. Part 3 will describe expanding the I/O operations to include parallel access to and from the Commodore 64, as well as adding a Centronix-type printer port to dump data for the disk directly to a parallel printer.

The User Command

The User Command ("**UA**" thru "**UJ**", or "**U1**" thru "**U:**") is provided to allow a machine language programmer to install custom commands and to define parameters for these commands. The user command takes the form:

```
"Ux" or "UX:" + CHR$(parameter 1)
  + CHR$(parameter 2) ...
```

In practice, the "x" is replaced by an index character which defines the location which is to be executed. Table 1 defines the valid index characters and their associated execution addresses in the 1541 memory.

Table 1

Index Character	Address Executed
A	$CD5F
B	$CD97
C	$0500
D	$0503
E	$0506
F	$0509
G	$050C
H	$050F
I	$FF01
J	$EAA0

Location $CD5F is the entry point for the read block routine. Location $CD97 is the entry point for the write block routine. Locations $0500 thru $0511 constitute the user jump table. The jump table is provided by the user and typically contains a three byte JMP instruction for each index character. Each of these jump instructions, when executed, causes a jump to the entry point of a user-provided routine in order to process the custom command. Index character **I** causes a jump to $FF01, but this has no practical value. Index character **J** causes a jump to $EAA0 which is the entry point for the power-up reset routine.

Parameters such as track number, sector number, byte count and pointers may accompany the user command. Parameter values are one byte,so they must be between 0 and 255, inclusive. Consider the following example:

```
"UD:" + CHR$(parameter 1)
  + CHR$(parameter 2) + ...
```

Parameter 1 will be located at address $0203, parameter 2 will be located at address $0204, and so forth. Up to 38 parameters may be used. The parameters may be read by the user-provided routine at the locations listed in Table 2.

Table 2

PARAM	LOCATION	PARAM	LOCATION
1	$0203	20	$0216
2	$0204	21	$0217
3	$0205	22	$0218
4	$0206	23	$0219
5	$0207	24	$021A
6	$0208	25	$021B
7	$0209	26	$021C
8	$020A	27	$021D
9	$020B	28	$021E
10	$020C	29	$021F
11	$020D	30	$0220
12	$020E	31	$0221
13	$020F	32	$0222
14	$0210	33	$0223
15	$0211	34	$0224
16	$0212	35	$0225
17	$0213	36	$0226
18	$0214	37	$0227
19	$0215	38	$0228

Biography

Mike is an electronics technician, technical writer, inventor of electronics devices, and designer of video games. After experiencing an electronic failure on his personal 1541 disk drive, he discovered that a lack of serious documentation for this device. He is author of the *1541 Single Drive Floppy Disk Maintenance Manual*, published by Peltier Industries, 1984.

Expanding the User Command

The user command may be expanded to include index characters **A** thru **P**. This gives the programmer 16 new commands to work with. Locations $006B and $006C, respectively, contain the low and high bytes of the starting location of the user address table. The user address table differs from the user jump table discussed earlier in that the user address table has only two bytes per index character. These bytes are the low and high bytes of the locations to be executed. In the unexpanded mode, $006B contains $EA and $006C contains $FF. These addresses point to the user address table starting at $FFEA. The user address table contains the following information:

Table 3

Address	Contents	Index	Vector
$FFEA	$5F	A	$CD5F
$FFEB	$CD		
$FFEC	$97	B	$CD97
$FFED	$CD		
$FFEE	$00	C	$0500
$FFEF	$05		
$FFF0	$03	D	$0503
$FFF1	$05		
$FFF2	$06	E	$0506
$FFF3	$05		
$FFF4	$09	F	$0509
$FFF5	$05		
$FFF6	$0C	G	$050C
$FFF7	$05		
$FFF8	$0F	H	$050F
$FFF9	$05		
$FFFA	$01	I	$FF01
$FFFB	$FF		
$FFFC	$A0	J	$EAA0
$FFFD	$EA		

Notice that locations $FFEE thru $FFF9 in the user address table contain vectors which point to the user jump table in locations $0500 thru $0511. Although only 10 index characters are shown in the table, DOS will support up to 16, giving a total table length of 32 bytes (two bytes per command). To expand the user command to include all 16 index character (**A** thru **P**), change the user address table pointer at $006B (low byte) and $006C (high byte) to point to a 32 byte USER table in RAM. Enter the table in the following format:

Table 4

Address	Contents	Index
p+0	$5F	A
p+1	$CD	
p+2	$97	B
p+3	$CD	
p+4	low byte vector	C
p+5	high byte vector	
p+6	low byte vector	D
p+7	high byte vector	
.	.	.
.	.	.
.	.	.
p+30	low byte vector	P
p+31	high byte vector	

(p = user address table base pointer contained in $006B and $006C)

Note that it is necessary to set p+1 thru p+3 to the values shown in order to preserve the read block (**U1** or **UA**) and the write block (**U2** or **UB**) commands. If, however, the read and write block commands are not needed, then the above locations may be used for other vectors. Each of the vectors in the above table point to entry points of the user-provided routine. The new table in RAM may be created or altered by the user. As in the unexpanded mode, the expanded mode also allows the use of parameters 1 thru 38.

To return to the unexpanded mode, use the "**U0**" command. This command sets the user address table pointer back to $FFEA, which restores the original user table.

Using all 16 expanded user commands may be difficult due to the lack of useable RAM for programming. Up to 1K of RAM may be used for programming (buffers 0 thru 3, $0300-$06FF). However, as these buffers are filled with programs, they are no longer available for data transfer, thus reducing the number of file which may be opened at any one time. This problem will be solved in Part 2 of this article by expanding the 1541 RAM to 6K. This will be accomplished with a plug-in 4K RAM module that you can build yourself.

A Very Moving Message

Split Screen, Fine Scrolling and Interrupt Techniques Combined in a Useful Utility for the Commodore 64

by Ian Adam
Vancouver, British Columbia, Canada

Summary

Here's a program you can very quickly type into your Commodore 64. It will add to your computer's usefulness and show off some of its good features at the same time. The program runs a 'marquee-type' message across the screen of the computer. This feature allows you to use the C64 as part of an 'electronic bulletin board' application, either alone or in combination with other programs. The message and its configuration can be custom-tailored to suit your needs. The program takes advantage of the computer's fine-scrolling capabilities to move the message smoothly and evenly across any part of the screen.

The Moving Message

Here's another application for home or club ... the computer as a moving message display device. This is a flashy way to leave a message for others in your family, or you can use it to announce schedules or special events in a club environment. You can even add it in to other programs and use the message to give instructions, advice in an adventure game and so on, while the other program is running. This little machine-language routine will run a continuous marquee-type message across either the top or the bottom of the screen, or any other line that you choose. The message that it runs can be up to 255 characters long, enough to cover most typical applications. The message or the way it is displayed can be changed easily to suit your needs. About the only limitation is your imagination!

Using the program is very simple. All you need to do is type in the BASIC program for your computer, being sure to SAVE a copy, then run it. You will then be asked to specify where on the screen you want the message to appear and to enter it according to the instructions. That's all there is to it! Your message will appear as if by magic, sliding continuously across the screen. Once it is working, it will continue to do so until you stop it.

The program shows off a number of very interesting features of your computer which we'll look at in a moment. In particular, note that it runs on an 'interrupt' basis. This means that it will continue to run when the program that loaded it is finished. You can even load in and run a different program and the message will still be displayed in most cases. The simplest way to stop the message is to press the RUN/STOP and RESTORE keys simultaneously. It can be restarted by typing SYS 49152.

For those who are interested in the details, the BASIC program loads in a machine-language program that does the serious work. This program is located in the spare RAM beginning at 49152. A commented assembly listing of the machine code is included in case you want to see how it works, but you'll still need the BASIC program to load in your message.

Additional Instructions

Once it is debugged and working, here are some additional instructions you may find useful in operating the display.

1. Changing text color: simply POKE location 49248 with a number from 0 to 15 to change the color of the text in the banner.
2. Background color: the color of the screen background and display background cannot be changed in the usual way (with a POKE to location 53281). The new locations to POKE are 49267 for the screen color and 49266 for the background of the message.
3. Changing the message: type GOSUB 9750 and follow the prompts to enter a new message. (alternatively, you can

```
  1Ø PRINT CHR$(147)
  2Ø PRINT " MOVING MESSAGE 64:
  3Ø PRINT " ──────────────────
  4Ø PRINT " BY IAN ADAM{DOWN}
  5Ø PRINT "THIS PROGRAM SCROLLS A BANNER MESSAGE
     ACROSS THE SCREEN USING
  8Ø GOSUB 95ØØ
  9Ø PRINT "{DOWN}
     *PRESS RETURN TO LOAD MESSAGE.{UP}":INPUT A$
 1ØØ GOSUB 96ØØ:GOSUB 975Ø
 11Ø PRINT "{DOWN4}ADJUSTMENTS:{DOWN}
 12Ø PRINT "POKE 49248, TEXT COLOUR
 13Ø PRINT "POKE 49266, COLOUR OF BANNER
 14Ø PRINT "POKE 49267, COLOUR OF MAIN SCREEN
 15Ø PRINT "SYS  49152  ENABLE MESSAGE
 16Ø PRINT "GOSUB 96ØØ  CHANGE MESSAGE LOCATION
 17Ø PRINT "GOSUB 975Ø  NEW MESSAGE
 18Ø END
9485 REM  READ AND POKE DATA INTO MEMORY
9500 FOR I=49152 TO 49273
9510 READ A:POKE I,A
9520 T=T+A:NEXT
9530 IF T-15669 THEN PRINT "CHECKSUM ERROR -
     DOUBLE-CHECK DATA!":STOP
9540 RETURN
9580 REM  WHICH LINE ?
9600 PRINT "{DOWN}PICK ANY LINE FOR THE MESSAGE
9610 PRINT "1 IS TOP LINE, 25 IS BOTTOM LINE
9620 INPUT "WHICH LINE";N%
9630 IF N%< 1 OR N%> 25 THEN 9610
9640 X=40*N%+768
9650 Y=X AND 255
9660 POKE 49245,Y:POKE 49250,Y
9670 X=(X-Y)/256
9680 POKE 49246,X:POKE 49251,X+212
9690 X=42+8*N%
9700 POKE 49271,X:POKE 49270,X+8
9710 RETURN
9730 REM  ENTER MESSAGE
9750 A$=CHR$(164):PRINT CHR$(147);
9760 FOR I=1 TO 255:PRINT A$;:NEXT
9770 PRINT CHR$(215)
9780 PRINT "{DOWN}
     TYPE MESSAGE AT TOP OF SCREEN, USING
9790 PRINT "LOTS OF SPACE, MOVE CURSOR DOWN TO {↑W}
     ,   THEN PRESS RETURN
9800 POKE 631,19:POKE 198,1
9810 INPUT A$
9850 A=49273:B=1023
9860 FOR I=1 TO 255:S=PEEK(B+I)
9870 IF S=100 THEN S=32
9880 POKE A+I,S:NEXT
9890 POKE 53265,27:SYS 49152:RETURN
9910 REM  ENTER DATA CAREFULLY !!!
9930 DATA 120,169,127,141,13,220,169,1,141,26,208,
     169,23,141,20,3,169,192
9940 DATA 141,21,3,88,96,173,25,208,141,25,208,162,0,
     189,114,192,141,33
9950 DATA 208,189,116,192,141,22,208,189,118,192,141,
     18,208,138,73,1,141,30
9960 DATA 192,240,3,76,188,254,165,162,41,7,73,7,141,
     116,192,201,7,208
9970 DATA 38,174,120,192,232,236,121,192,208,2,162,0,
     142,120,192,160,216,189
9980 DATA 122,192,153,232,6,169,15,153,232,218,232,
     236,121,192,208,2,162,0
9990 DATA 200,208,234,76,49,234,0,6,7,200,8,241,0,
     255
```

enter the message by poking the screen codes into memory beginning at location 49274].

4. Changing the location on the screen: type GOSUB 9600 and follow the prompts to change the location on the screen. Note that if you have not selected the top of the screen for the message, then scrolling the screen will create a bit of a mess.

5. Spacing: the moving message will have greater impact if you spread it out a bit, adding lots of spaces or other characters such as asterisks. The full length of the message is normally displayed, taking 34 seconds to scroll across the screen of the C64. The message will then repeat indefinitely. It is normally best to operate the program that way, but if you have a short, urgent message that you want to repeat more often, you can POKE its length into location 49273. If you want to display a very long message, your program can simply divide the message into segments of 255 characters and change the segment occasionally.

Technical Features

This program demonstrates a number of the special features of your Commodore 64. If you're not 'into' machine language or details of the computer you can just skip over this section, since you don't need to know all the details in order to run the program. However, if you want to know a bit of what makes the machine tick, then read on.

The BASIC program you see is a 'loader' for a machine language routine that, in effect, becomes part of the computer's operating system. This approach shows off at least three of your computer's special features:

1. modifying the interrupt routine.
2. split screen techniques.
3. fine scrolling and other control registers on the video chip.

Here's a little more on how each of these works:

Interrupt routine

Sixty times each second, the computer 'interrupts' what it's doing to carry out some housekeeping chores - checking the keyboard, updating the clock, washing the dishes and so on. In order to do this, it jumps to a special routine in memory that contains the instructions. The computer needs to know where this 'interrupt routine' is

located, so it stores the address in a pointer, held in locations $0314 and 0315 (decimal numbers 788 and 789).

The address normally held in that pointer is $EA31 on Commodore 64. What we will do is change the pointer so that it directs the computer to our routine instead. Our routine will take care of moving the message one step across the screen; when it's finished, we return control to the normal routine, to take care of the usual housekeeping tasks.

Split screen technique

The picture on your TV or monitor is formed by a series of horizontal scan lines projected sequentially from top to bottom of the screen. The video chip in the computer keeps track of where the raster scan line is on the screen and will alert you at any point on the screen you request. This is handy for switching various displays in and out on different parts of the screen, resulting in a display that could be, for example, part graphics and part text. This is known as a split screen.

In this case, we will ask the chip to generate an interrupt twice on each screen: first, when the scan line gets to the beginning of our message, we will select the color of the screen to suit. We will also adjust the horizontal position and narrow the screen to hide the letters coming on. Then, when the video chip indicates it has reached the bottom of our message line, we will set the screen color, position and width back to normal. We will also take this opportunity to slip the display a little to the left, while it is out of sight (so that it won't flicker).

Fine scrolling

It is fairly easy to write a program like this to jump the message across the screen one letter at a time - in fact, you can even do it using BASIC. However, this program uses the capability of the C64 to move the display across the screen one pixel at a time. This results in a very smooth scrolling effect, as there are eight pixels to each character. You have no doubt seen this feature used before, although you may not have realized it. It is often used by games to create an illusion of horizontal motion on part of the screen, for example to move traffic from side to side in frog games, or to scroll helicopter battles sideways.

```
                ;-------------------------------
                ;   MOVING MESSAGE PROGRAM
                ; FOR USE ON THE COMMODORE 64
                ;-------------------------------
                ; BY IAN ADAM
                ;
                ; SYS 49152 TO ENABLE THE
                ; MOVING MESSAGE DISPLAY.
                ;
00A2            CLOCK   EQU $A2
0314            IRQPTR  EQU $0314
0328            SCRN    EQU $0328
D012            VICRST  EQU $D012
D016            VICCTL  EQU $D016
D019            VICIRQ  EQU $D019
D01A            VICIMR  EQU $D01A
D021            VICBGC  EQU $D021
DC0D            CIAICR  EQU $DC0D
D728            SCOL    EQU $D728
EA31            OLDIRQ  EQU $EA31
FEBC            CLNUP   EQU $FEBC
                ;
C000                    ORG $C000     ; ROUTINE RESIDES IN UPPER RAM
                ;
C000 78         SETUP   SEI           ; BLOCK OUT INTERRUPTS
C001 A9 7F              LDA #$7F
C003 8D 0D DC           STA CIAICR    ; TURN OFF HARDWARE TIMER
C006 A9 01              LDA #$01
C008 8D 1A D0           STA VICIMR    ; ENABLE RASTER INTERRUPTS
C00B A9 17              LDA #MOVER
C00D 8D 14 03           STA IRQPTR    ; RESET IRQ POINTER TO START
C010 A9 C0              LDA /MOVER    ; OF NEW PROGRAM
C012 8D 15 03           STA IRQPTR+1
C015 58                 CLI           ; RE-ENABLE INTERRUPTS
C016 60                 RTS           ; RETURN TO BASIC
                ;
C017 AD 19 D0   MOVER   LDA VICIRQ    ; ACK INTERRUPT TO
C01A 8D 19 D0           STA VICIRQ    ; VIDEO CHIP
C01D A2 00      PNTER   LDX #$00      ; POINTER TO BYTE +1
C01F BD 72 C0           LDA COLOR,X   ; GET BACKGROUND COLOR
C022 8D 21 D0           STA VICBGC    ; AND STORE IT
C025 BD 74 C0           LDA CONFG,X   ; GET SCREEN CONFIG.
C028 8D 16 D0           STA VICCTL    ; AND STORE IT
C02B BD 76 C0           LDA RASTR,X   ; GET NEXT RASTER VALUE
C02E 8D 12 D0           STA VICRST    ; AND STORE IT
C031 8A                 TXA           ; LOOK AT POINTER
C032 49 01              EOR #$01      ; FLIP IT OVER
C034 8D 1E C0           STA PNTER+1   ; AND STORE FOR NEXT TIME
C037 F0 03              BEQ SLIDE
C039 4C BC FE           JMP CLNUP     ; IF MESSAGE NO ON, GET OUT.
                ;
                ; IF THE MESSAGE IS NOT BEING SCANNED,
                ; RELOCATE IT SO THAT THE DISPLAY IS
                ; NOT JERKY.
                ;
C03C A5 A2      SLIDE   LDA CLOCK     ; SET SYSTEM CLOCK BYTE
C03E 29 07              AND #$07      ; CONVERT INTO DECREASING SCAL
C040 49 07              EOR #$07      ; FOR A SMOOTH SCROLL
C042 8D 74 C0           STA CONFG     ; SAVE FOR NEXT TIME
C045 C9 07              CMP #$07      ; IF 7, THEN SCROLL MESSAGE LE
C047 D0 26              BNE EXIT      ; IF NOT 7, BYPASS THESCROLL
                ;
C049 AE 78 C0   SCROLL  LDX POSITN    ; WHERE IN THE MESSAGE?
C04C E8                 INX           ; MOVE OVER ONE SPACE
C04D EC 79 C0           CPX LENGTH    ; ARE WE AT THE END ?
C050 D0 02              BNE CONTIN    ; NO, SO CONTINUE
C052 A2 00              LDX #$00      ; YES, SO START OVER
C054 8E 78 C0   CONTIN  STX POSITN    ; SAVE FOR NEXT TIME
C057 A0 D8              LDY #$D8      ; Y REGISTER IS SCREEN INDEX
                ;
```

```
C059 BD 7A C0   GETCHR   LDA MESAG,X   ; GET A LETTER OF MESSAGE
C05C 99 28 03            STA SCRN,Y    ; AND DISPLAY IT ON SCREEN
C05F A9 0F               LDA #$0F
C061 99 28 D7            STA SCOL,Y    ; POKE COLOR OF TEXT
                ;
                ; SCRN AND SCOL ARE OFFSET VALUES FOR
                ; START OF SCREEN DISPLAY AND COLOR
                ; MEMORY. THEY ARE CHANGE BY THE BASIC
                ; LOADER IF NECESSARY.
                ;
C064 E8                  INX
C065 EC 79 C0            CPX LENGTH    ; END OF MESSAGE ?
C068 D0 02               BNE NEXT      ; NO
C06A A2 00               LDX #$00      ; YES, CONTINUE FROM START
                ;
C06C C8         NEXT     INY           ; NEXT SCREEN POSITION
C06D D0 EA               BNE GETCHR    ; MORE TO DO?
C06F 4C 31 EA   EXIT     JMP OLDIRQ    ; OVER AND OUT
                ;
                ; FINISH CHORES BY JUMPING BACK TO
                ; THE STANDARD 64 IRQ ROUTINE
                ;
C072 00 00      COLOR    DBY 0
C074 00 00      CONFG    DBY 0
C076 00 00      RASTR    DBY 0
C078 00         POSITN   BYT 0
C079 00         LENGTH   BYT 0
C07A 00         MESAG    BYT 0         ; START OF MESSAGE AREA
C07B                     END
```

One Final Message

Try this program in your machine - I'm sure you'll find it useful and it gives a few more ways to put your powerful home computer to practical use. I'll award a prize for the most imaginative application I hear of. Meanwhile, may the power be with you!

Biography

Ian Adam is a Transportation Engineer with the City of Vancouver, British Columbia. Home to mountains, killer whales, and lots of rain, British Columbia is known locally as 'Lotus Land'. After dealing with the routines of traffic and transit through the city all day, Ian finds routing information around the inside of a computer an ideal way to relax. While also programming on the VIC 20, Apple II and IBM PC, his preference is for the Commodore 64. Unfortunately, he must share this with his wife Linda and his two sons Paul and Doug.

feedback

Dear Harvey,

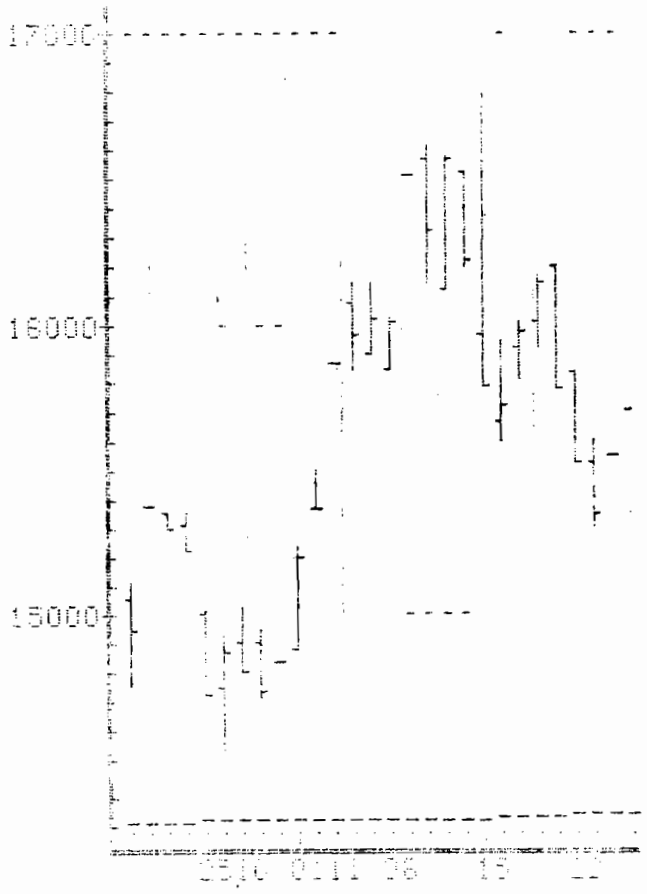
In response to your letter in MICRO, June 1984, I offer the following.

First, with the exception of addition of a second time series and third label, the Quickplot(tm) routine which is part of Quicktrieve supplied by Commodity Systems Inc., 200 W. Palmetto Park Road, Boca Raton, FL 33432 (305/392-8663), does what you want with either daily commodity futures or stock or stock options data which they sell. Unfortunately, the program is supplied only in compiled form and is intimately tied to acquisition through their system on a regular basis.

I expect to eventually write my own program of that nature, if the demand develops. In the meantime, I do my plots on an MX-100 with substantially higher resolution than Apple Hi-Res graphics. A sample is enclosed. I have several years of use of Apple in commodity trading, If this interests you, or any MICRO reader, feel free to write.

Jere Murray
Seldovia Paint Software
Box 237
Seldovia, AK 99663

Portion of Sample Commodity Chart
(The whole chart is 15 by 25)



Dear Mr. Tripp:

I'm delighted to find the lead article on the Dvorak keyboard in your July 1984 issue. There is, indeed, a growing awareness of the inherent awkwardness of the QWERTY keyboard arrangement.

After 20 years of QWERTY in my daily work, I switched to Dvorak in 1965, cold turkey. I was never sorry, not even one day, though for the first six weeks or so my usual 80 wpm began again from 15 to 20 wpm and slowly became 35, then 60, and a year later 100 plus. I cannot imagine going back, although my two daughters, trained from high school on Dvorak, had to switch to QWERTY in order to obtain employment in their chosen line of work. Those were pre-microcomputer years. Now, there's no problem getting the new ANSI Standard Keyboard (ASK, which is Dvorak with the numeral row left in the old ascending arrangement) on virtually any computer. The article neglected to mention the 1983 action of the American National Standards Institute in adopting Dvorak's ASK variant officially. Many persons might take Dvorak more seriously once they realize that it has official blessing.

It is not even mentioned by Radio shack, more's the pity!, that SCRIPSIT has the ASK keyboard arrangement resident. For those who use SCRIPSIT 2.0, a simple patch on a working master is all that's required to convert their keyboard (from TRSDOS Ready, PATCH SCRIPSIT ADDFA FD7 CD8). Thus, Radio Shack is among the very first computer manufacturers, along with IBM and Apple, to recognize the value of the new keyboard, in spite of their apparent modesty. Apple IIc has a hardware switch for using either keyboard at will.

The DSK is everything it's touted to be, and more. There are a growing number of converts to ASK. But since DSK and ASK are in the public domain, why try to persuade anyone that there's an advantage in switching away from QWERTY, since there's no money to be made? It's a rare person who believes that a gift can come sans strings attached. I can use QWERTY at 35 wpm, DSK and ASK as 100. It is not easy to learn to use more than one of them, since there is a tendency to overlap, just as a foreign language user slips an occasional native word into his conversation. But it does give me a basis for objective evaluation of all three: I pronounce Dvorak superlative in all respects. ASK is a beautiful compromise.

Sincerely,

Waldo T. Boyd
P.O. Box 86
Geyserville, CA 95441

by Ralph Tenny
Richardson, Texas

A Mystery !

At first glance, the Commodore 64 seems to be ideally suited for timing and counting measurements. That is, it has two Time of Day clocks and two independent counters for our use, and all of them produce interrupts. However, I developed a number of frustrations while trying to use these features to calibrate a Voltage-to-Frequency Converter (VFC). Let's get some background on VDCs and their calibration; then I'll explain my frustrations. If any one of you can solve the problems more neatly than I did, the podium is yours! Just write me c/o MICRO and tell me your solution.

Figure 1 shows the circuit of the VFC as implemented on the User Port adapter reported several columns back (MICRO 70:54, March 1984). (For those who haven't seen that report, skip forward to the section on the User Port Adapter.) U1 is the VFC, which requires + and - 5V power supplies. U2 is a low-power inverter which runs from +5V and produces about -4.5 V. Only a few connections are made to the User Port for this circuit: +5V, Common, PB7 and CNT2. Depending on the operating mode, you will count pulses or the time between two pulses, depending on the calibration mode.

VFCs Explained!

Most modern VFCs work on a *charge balancing* principle which minimizes error producing influences. Charge balancing is done by charging a capacitor from an external source for a while, then quickly discharging it from an internal calibrated source. The internal discharge rate is constant, and the external charging rate is proportional to the external voltage. (The length of time for external charging is set by the external voltage.) With a large external voltage, the capacitor is charged rapidly, so the switch-over to discharge comes quickly. The switch signal (see Figure 2a) is brought out for our use, and its frequency is directly proportional to the external voltage. The Teledyne 9400 VFC used in this experiment also furnishes a square wave at half the output frequency (Figure 2b). We will use both in the experiment to be discussed.

A number of very good VFCs are available at quite low prices. Typical specifications are: 0.01% linearity to 10 KHz and 0.0025% gain stability with temperature. At full scale (10 V input = 10 KHz), the measurement resolution is better than 13 bits! A 13-bit A/D converter is several times more expensive, depending on how fast the conversion is. Even at 1 V input (1 KHz), the resolution is almost 10 bits. The tradeoff is that full-resolution measurements require one second to complete. Also, the measurements

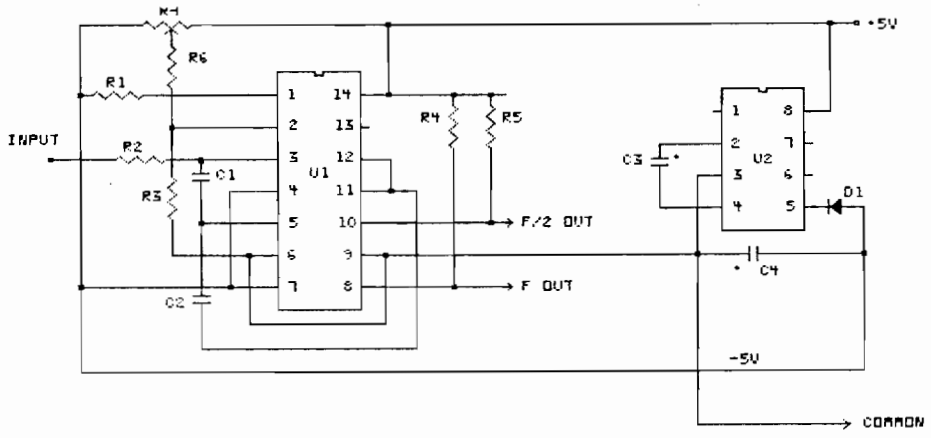


Figure 1

VFC Analog/Digital Converter Circuit

Parts List for VFC

R1 - 100 Kohm, Carbon Film, 5%, 1/4 watt
R2 - 1 Megohm, Metal Film, 1%, 1/4 watt
R3 - 10 Kohm, Carbon Film, 5%, 1/4 watt
R4, R5 - 8.2 Kohm, Carbon Film, 5%, 1/4 watt
R6 - 510 Kohn, Metal Film, 1%, 1/4 watt
R7 - 50 Kohm, Cermet, single turn, 1/8 or 1/4 watt
C1 - 230 pF, mica or NPO ceramic capacitor
C2 - 1000 pF, mica or NPO ceramic capacitor
C3 - 10 uF Tantalum Dipped electrolytic capacitor
C4 - 100 uF Aluminum electrolytic capacitor
U1 - Teledyne 9400 VFC
U2 - Intersil ICL7660 DC Converter

need two bytes of memory for storage of each test result. If you don't need high resolution, use one-tenth second measurement periods to save time and memory space.

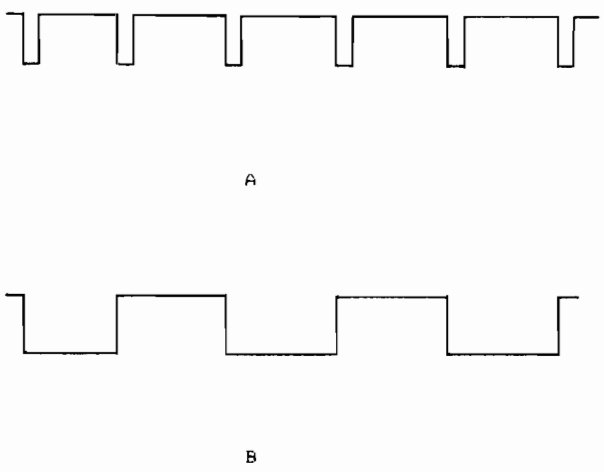


Figure 2 Output Waveforms from the VFC

Calibration Woes

Two calibration points are needed on this particular VFC. First, I need to set the frequency output for 10 KHz with 10 V input, and then minimize the zero offset. Zero offset affects the accuracy at very low input voltage, so the frequency is adjusted to 20 Hz at 20 mV input. This calibration must use a period measurement to get adequate resolution.

My first frustration came when trying to calibrate the full-scale response of this VFC. I have been using HESMON64 for all of my hardware and interfacing experiments, with mostly good results. The first idea I had for full-scale calibration was to operate the TOD clock on U2 (NMI interrupt) with one second interrupts. None of the several reference books I have mentions how to set the TOD interrupts, except in a general way. *Setting* the TOD clock is supposed to be done beginning with the Hours/AM-PM register and finishing with the Tenth Second register. The clock starts running when the Tenth Second register is loaded, so that the starting time can be precisely controlled. Setting the Alarm is supposed to be the same, except that BIT7 of each register is supposed to be set also.

The bit map of the TOD registers looks like this:

ADDR	REG	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
DD08	SEC/10	0	0	0	0	T8	T4	T2	T1
DD09	SEC	0	SH4	SH2	SH1	SL8	SL4	SL2	SL1
DD0A	MIN	0	MH4	MH2	MH1	ML8	ML4	ML2	ML1
DD0B	HRS	PM	0	0	HH1	HL8	HL4	HL2	HL1

Note that the registers are arranged in ascending order, with time kept in BCD. The first hurdle is that HESMON reads and writes eight bytes in its memory modifying mode, reading from low to high. So, if you manage to get the clock running with HESMON, reading the time reads the Hours register last, stopping the clock. Similarly, writing via memory modification writes the Tenth Second register first instead of last, failing to start the clock. The clock can be started using this brief program:

```
A2 03             LDX #$03      ; MOVE FOUR NUMBERS
BD 0C C0   GET    LDA BUFR,X    ; GET DATA
9D 08 DD          STA $DD08,X   ; AND WRITE IT
CA                DEX           ; IN REVERSE ORDER
10 F7             BPL GET       ; LOOP
00                BRK           ; HESMON STOP
0A 0F 1E   BUFR   BYT HRS,MIN,SEC,STEN
         ; SAMPLE SETTINGS
           HRS    EQU 10
           MIN    EQU 15
           SEC    EQU 30
           STEN   EQU 06
```

The clock starts easily with that program segment. Using several similar segments to set the ALARM function apparently failed each time. At least, I couldn't find an interrupt service routine which demonstrated interrupt operation!

My intent had been to use one-second interrupts from the TOD clock to calibrate the VFC. In this operation, the VFC pulse output is connected to CNT2 so that Timer B accumulates pulses from the VFC. Eventually, I devised this program to generate one second interrupts for that purpose:

```
CE 00 20          DEC $2000     ; REP COUNTER
10 09             BPL KI        ; RESET INTERRUPT
A9 00             LDA #00       ; STOP TIMERS
8D 0F DD          STA $DD0F     ; TIMER B
8D 0E DD          STA $DD0E     ; TIMER A
00                BRK           ; RETURN TO HESMON
AD 0D DD   KI     LDA $DD0D     ; RESET INTERRUPT
40                RTI           ; END INTERRUPT SERVICE
A9 D5             LDA #$D5      ; SET TIMER A
8D 04 DD          STA $DD04     ;  FOR 0.1 SECONDS
A9 27             LDA #$27
8D 05 DD          STA $DD05
A9 09             LDA #$09      ; NEED 10 TIMER
8D 00 20          STA $2000
A9 FF             LDA #$FF      ; FULL COUNT IN TIMER B
8D 06 DD          STA $DD06
8D 07 DD          STA $DD07
A9 01             LDA #$01      ; SET NMI* VECTOR
8D 18 03          STA $0318
A9 20             LDA #$20
8D 19 03          STA $0319
A9 81             LDA #$81      ; ENABLE TIMER A
8D 0D DD          STA $DD0D     ; INTERRUPT
A9 09             LDA #$09      ; START TIMER B
8D 0F DD          STA $DD0F     ;  IN ONE-SHOT MODE
A9 01             LDA #$01      ; START TIMER A
8D 0E DD          STA $DD0E     ;  IN FREE-RUN MODE
4C 42 C0   LP     JMP LP        ; TIGHT LOOPS 1
```

When you RUN this program it should come back with the HESMON prompt almost immediately. Examine memory at $DD06-DD07 (count in Timer B registers). The recorded count with +5V input will probably be about $1388. There are two ways to calibrate this circuit. The first is to input a precise voltage (such as 10.0V) and adjust the value of R1 so the count is exactly $2710 (10,000 decimal). The second is to put in a known voltage near full scale and record the count. From these values you can then compute a counts/volt and compute other voltages using this value. Here's an example:

9.3V input yields $244A or 9290 decimal.
9290/9.3 = 998.9 counts/volt.

If you get a count of 6320 decimal, the voltage is 6320/998.9 or 6.39V. The computations above were rounded off somewhat. It is unrealistic to keep all the digits your calculator gives you!. If you can calibrate this circuit to 0.5%, three digit resolution can exceed the accuracy available to you, depending on the input voltage.

Most analog devices have zero offset, and VFCs are no exception. The zero offset calibration experiment was a really major frustration. The calibration should be done with 20 mV input (output 20 Hz). By measuring the period of the VFC output with a timer counting the processor clock, significant resolution is possible. My original plan was to drive the FLAG input with the VFC output. FLAG is an edge-sensitive input, which means that once the line has been driven low it must go high before another interrupt can be generated. Also, the first interrupt must be cleared before another interrupt will be issued. (Clearing any interrupt of the 6526 CIA is accomplished by reading the Interrupt Control Register.) To make a bitter story short, there apparently was severe interaction between *any* program using FLAG and the HESMON cartridge. The program below reads the half-period of the F/2 output. it waits until F/2 goes high, then low, and starts the counter. When F/2 goes high again, the counter is stopped. Connect the F/2 output to BIT6 of the User Port. Enter this program and run it repeatedly with a .02 V input to the VFC. Check the Timer B count after each run and adjust R7 until you get 51020 counts ($C74C). Note: since the timer starts at $FFFF and counts down, $FFFF - $C74C = 38B3. That is, the counter will show $38B3 when the adjustment is correct.

```
78                  SEI          ; DISABLE IRQ*
A2 ØF               LDX #$ØF     ; COUNTER START
AØ ØØ               LDY #$ØØ     ; COUNTER STOP
A9 FF               LDA #$FF     ; SET COUNTER B FOR
8D Ø6 DD            STA $DDØ6    ; MAXIMUM COUNT
8D Ø7 DD            STA $DDØ7
2C Ø1 DD   IN1      BIT $DDØ1    ; TEST FOR F/2 LOW
5Ø FB               BVC IN1      ; LOOP IF LOW
2C Ø1 DD   IN2      BIT $DDØ1    ; TEST FOR HIGH
7Ø FB               BVS IN2      ; LOOP WHILE HIGH
8E ØF DD            STX $DDØF    ; START COUNTER
2C Ø1 DD   IN3      BIT $DDØ1    ; TEST FOR LOW
5Ø FB               BVC IN3      ; WAIT FOR LOW TO STOP
8C ØF DD            STY $DDØF    ; THEN TURN OFF COUNTER
ØØ                  BRK          ; BACK TO HESMONS1
```

It was particularly irritating to have to resort to software loops as the *only* available way to make this calibration measurement! Remember: If you can make Commodore user interrupts work well with HESMON or another debugger, tell us how!

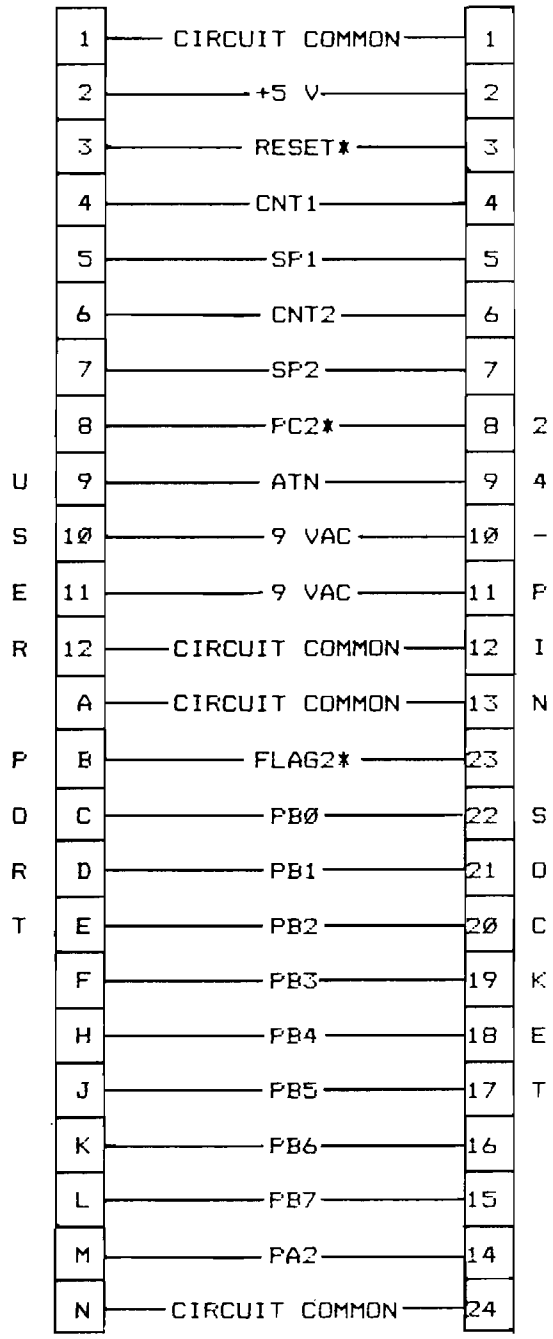


Figure 3

Wiring Chart for User Port Adapter

You should be aware of some constraints on using this type of program for calibration against the processor clock. I tested that program with 5 V input to the VFC (about 5 KHz output). Without the SEI instruction, internal interrupts would scramble about one sample in four. Even with the SEI, one in seven samples were stretched; I have no explanation. At the *real* calibration point, (20 Hz output), the signal was somewhat variable. Pulse jitter due to noise pickup was visible on an oscilloscope, but not to the extent shown in the data. The final calibration was done using an average of ten samples. The observed jitter in the data was ±2%.

User Port Adapter

Anytime you interface to a computer, it is much easier if you have a special work area for your experiments. The User Port is on the left rear of the Commodore 64, with a place to plug on an edge-card connector. I developed a simple adapter which extends the User Port around near the front of the computer. Figure 3 shows the circuitry between the User Port and the extender cable. This adapter is simply a short piece of experimenter perf-board with a dual-readout 12 position connector soldered to it. The connector fits the User Port, and a 24 pin IC socket is mounted on the perf-board. A flat cable with 24 pin headers on each end carries the User Port connections to the front of the C-64. This end of the flat cable plugs into any of several breadboards and permanent boards which are experimental circuits or permanent interfaces.

Future Projects

For those who might be following the design discussion of networking my three computers, I'm still planning to report on the bus loading of the interface detailed in the previous column, and discuss a full eight-bit wide I/O port to replace the four-bit ports shown last month. The interface project has been temporarily slowed to meet other deadlines, and to order some interface breadboard cards to build the interface on. When the interface project gets under way again [probably by next column], at least two such breadboard cards will be discussed and the parts layout shown. In addition, the support software will be outlined.

Updates & Microbes

BASIC Hex Loader, Micro #73, page 65:
Line 9 in listing 2 had a '-' inserted by the typesetter. It should read:

```
9 MS> ME THEN PRINT "XX";:I=6:J=11
```

The Hex Loader was written to work with any Microsoft-like BASIC. Since the Atari's BASIC is significantly different, particularly in the area of string manipulation, a special version is required. Here is the Hex Loader modified for the Atari.

```
1Ø DIM X$(4),HX$(5Ø)
11 READ X$:Z=LEN(X$):GOSUB 17:MS=X:Z=2
12 READ HX$:J=1
13 X$=HX$(J,J+1)
14 IF X$=XX THEN END
15 IF X$=YY THEN GOTO 12
16 GOSUB 17:POKE MS,X:MS=MS+1:J=J+2:GOTO 13
17 X=Ø:FOR I=1 TO Z:Y=ASC(X$(I,I)):
   IF Y> 57 THEN Y=Y-7
18 Y=Y-48:X=X*16+Y:NEXT I:RETURN
```

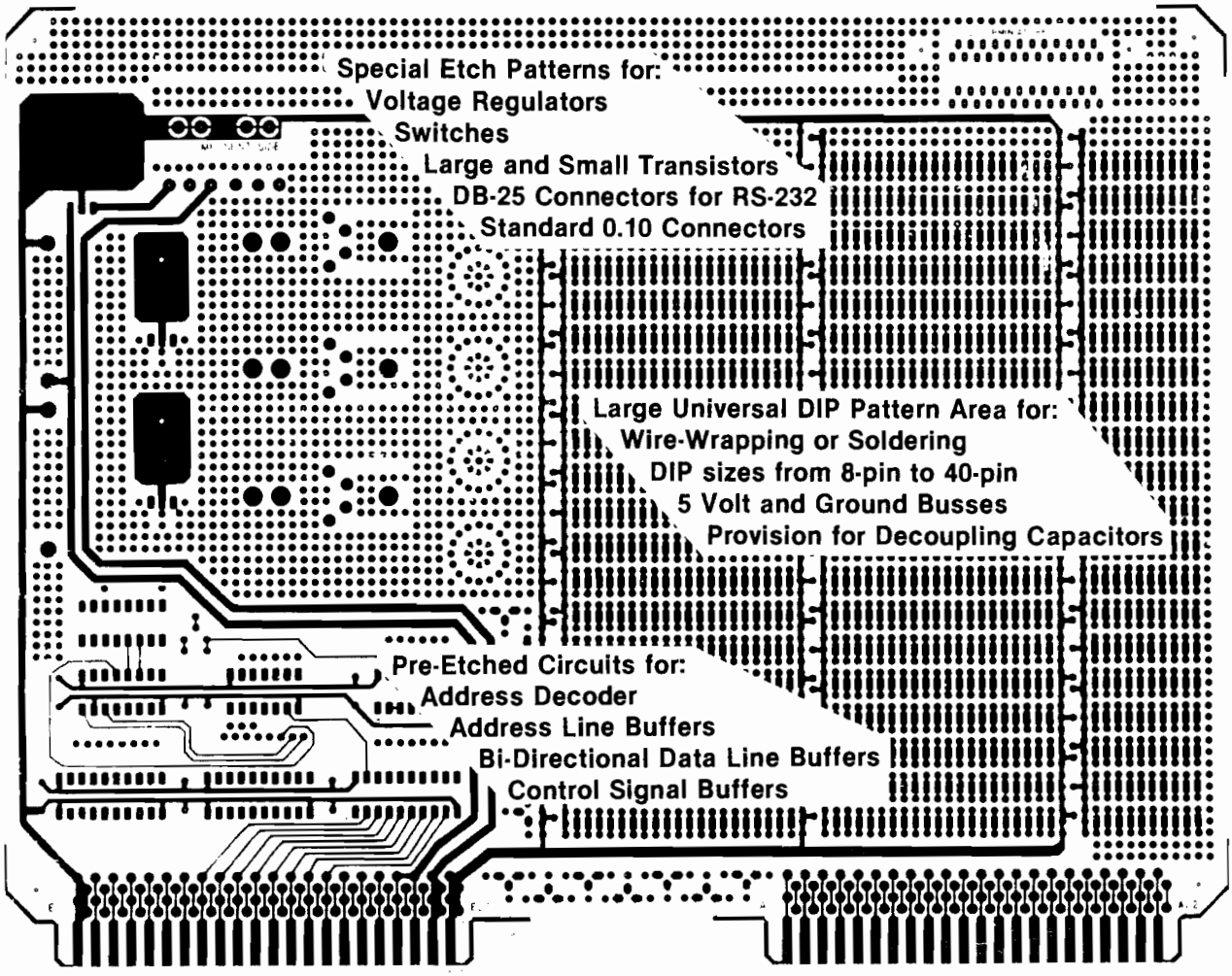
Sample DATA Statements

```
1ØØ DATA 6ØØ          Starting Address
1Ø1 DATA A57A8D7ØYY   Hex Data
```

8" by 10" Prototyping and Custom Circuit Board

Proto Plus

Professional Quality
Double-Sided with Gold Plated Edge Connectors
Address and Data Buffering
Address Decoding Circuitry
Patterns for Special Devices
Large 8 by 10 inch Work Area
Patterns for Soldering or Wire-Wrap



Hundreds of these Proto Plus boards were sold for $50 to $75 dollars. Since we are no longer actively in the PC board market, we are offering the remaining stock at a great discount.

Now Only $25.00 While Supplies Last.

The Ideal Board for the Interface Clinic and other MICRO Projects.

We have a limited supply of other boards — Video Plus, Dram Plus, Memory Plus, Flexi Plus, Keytronic keyboards and other "good stuff". No reasonable offer refused. For a complete list, send a self-addressed, stamped business-size envelope to: **Surplus List**

The Computerist, Inc.
P.O. Box 6502
Chelmsford, MA 01824

Quick Cipher Routine
Protecting Your Information

by Art Matheny
Lutz, Florida

Summary
In less than three seconds this 6502 machine language subroutine can encode 24 kilobytes into a cipher that is difficult to crack. The same routine decodes the data just as quickly.

How You Can Keep Your Information Private — Even While Using Public Communications Channels and Databases

Before storing a program or data on a public mass storage device or transmitting it over a public communications channel, it may be wise to encipher it. This short 6502 machine language subroutine quickly encodes a specified range of memory. The same subroutine decodes the data to recover the original memory contents. The routine takes less than three seconds (with a 1 MHz clock) to encode or decode 24 kilobytes. This is short compared to the time it usually takes to store or transmit the same amount of data.

The contents of the memory range could be a BASIC program, machine code, data tables, or a combination of all of these. it does not matter since every byte in the specified range undergoes transformation. Cryptographers call the original memory contents the plaintext, and they call the transformed data the ciphertext. The goal is to design a code such that if any codebreaker gets hold of the ciphertext, he would have a hard time recovering the plaintext.

The simplest scheme I have seen used for encryption of computer memory is as follows: One picks any number between 1 and 255 to use as a key. The ciphertext is produced by performing an exclusive-or (EOR) of this byte with every byte of the plaintext. To recover the plaintext, EOR the same key with every byte of the ciphertext. The trick is that two EOR instructions with the same byte

```
                ;******
                ;CIPHER
                ;******
                ;
                ;QUICK CIPHER ROUTINE
                ;BY ART MATHENY
                ;   TAMPA, FL
                ;
                ;ENCODES OR DECODES
                ;A RANGE OF MEMORY
                ;
                ;-------------------------------
                ;THE FOLLOWING 2-BYTE POINTER
                ;CAN BE LOCATED AT ANY
                ;CONVENIENT ZERO-PAGE ADDRESS
                ;-------------------------------
                ;
ØØFB             PTR      EQU $FB       ;Z-PAGE POINTER
                ;
                ;-------------------------------
                ;CHOOSE CONVENIENT ORIGIN.
                ;THE FOLLOWING IS THE START OF
                ;THE CASSETTE BUFFER FOR
                ;COMMODORE 64 OR VIC-2Ø.
                ;-------------------------------
                ;
Ø33C                      ORG $33C
                ;
                ;-------------------------------
                ;JUMP TABLE
                ;-------------------------------
                ;
Ø33C 4C 4A Ø3             JMP CIPHER
Ø33F 4C 7Ø Ø3             JMP RND
                ;
```

```
                        ;-------------------------------
                        ;ARGUMENTS
                        ;-------------------------------
                        ;
0342 12          SEED   BYT $12        ;POKE ANY 4-BYTE
0343 34                 BYT $34        ;SEED HERE.
0344 56                 BYT $56        ;USE SAME SEED AT
0345 78                 BYT $78        ;DECODING TIME
0346 00 08       FIRST  BYT 0,8        ;STARTING ADDR
0348 FF 67       LAST   BYT $FF,$67    ;FINAL ADDR
                        ;
                        ;-------------------------------
                        ;CIPHER
                        ;-------------------------------
                        ;
                        ;COPY STARTING ADDR TO Z-PAGE
                        ;
034A AD 46 03    CIPHER LDA FIRST
034D 85 FB              STA PTR
034F AD 47 03           LDA FIRST+1
0352 85 FC              STA PTR+1
                        ;
                        ;EOR MEMORY BYTE WITH RANDOM BYTE
                        ;
0354 A0 00              LDY #0
0356 20 70 03    CIP1   JSR RND
0359 51 FB              EOR (PTR),Y
035B 91 FB              STA (PTR),Y
                        ;
                        ;INCREMENT MEMORY POINTER
                        ;
035D E6 FB              INC PTR
035F D0 02              BNE CIP2
0361 E6 FC              INC PTR+1
                        ;
                        ;LAST BYTE YET
                        ;
0363 AD 48 03    CIP2   LDA LAST
0366 C5 FB              CMP PTR
0368 AD 49 03           LDA LAST+1
036B E5 FC              SBC PTR+1
036D B0 E7              BCS CIP1
036F 60                 RTS
                        ;
                        ;-------------------------------
                        ;RANDOM NUMBER GENERATOR
                        ;RETURNS RANDOM BYTE IN THE .A REGISTER
                        ;-------------------------------
                        ;
0370 18          RND    CLC
0371 A2 03              LDX #3
0373 BD 42 03           LDA SEED,X
0376 CA                 DEX
0377 7D 42 03    RND1   ADC SEED,X
037A 9D 42 03           STA SEED,X
037D CA                 DEX
037E 10 F7              BPL RND1
0380 A2 03              LDX #3
0382 FE 42 03    RND2   INC SEED,X
0385 D0 03              BNE RND3
0387 CA                 DEX
0388 10 F8              BPL RND2
038A 60          RND3   RTS
038B                    END
```

Art Matheny was a physics teacher before he took up full-time programming. He is now employed at the University of South Florida as assistant in scientific computing for the College of Natural Sciences. At home he programs on his Apple II, Commodore 64 and VIC-20.

has no net effect. This simple code disguises the data alright, but it is nothing more than a simple substitution code, which can be cracked by age-old methods. The code-breaker can determine the substitution table without ever having figured out that the EOR instruction was used.

My scheme also uses the EOR instruction, but first a fast random number routine generates values to EOR with the data. The key in this case is a four-byte SEED value, which determines the pseudo-random sequence. Decoding is accomplished by calling the same routine with the SEED bytes reset to the key values. At the start of encoding, any four bytes can be poked into the SEED locations. Because these bytes are modified by the program, they must be reset to the same key values at the start of decoding. The FIRST and LAST addresses of the memory range must also be set. All of these arguments are located immediately following the jump table at the start of the program.

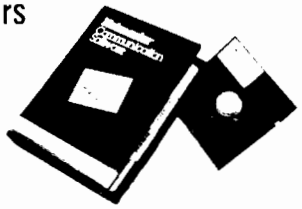
You do not have to use the particular random number generator given here. Any routine that generates one-byte values will work. The one included here was originally published in MICRO #51 (August 1982). In this application, its main virtues are that it is fast (about 69 cycles) compared to more sophisticated routines and that it generates an extremely long non-repeating sequence. You may even choose to invent your own personal number generator. It does not have to be perfectly random, rather use some scheme that is so unlikely that the code-breaker would fail to guess it. Since many numbers need to be generated, speed is important. The main point is that the number sequence must be deterministic, while not appearing so. The sequence should be determined by some a seed value, which functions as the key of the code. The random number generator must return a random byte in the .A register.

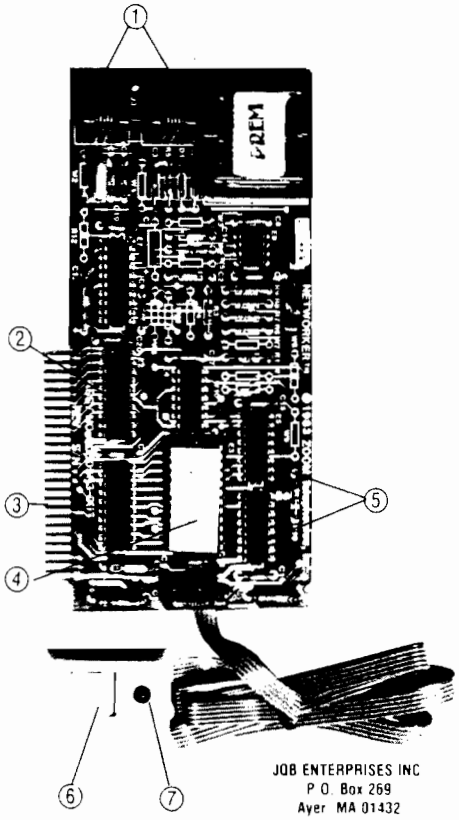
All that is required to use the routine, once you have it loaded is to set the seed bytes and the starting and ending address of the information to be encoded. The addresses are stored in normal low/high byte order (at $0346 and $0348 in this version). Then, make a subroutine call to CIPHER through its jump table vector (JSR $33C) from an assembly program or a SYS call from BASIC (SYS 828). Your encoded/decoded information will replace your original information.

FREE! NETWORKER NOW INCLUDES A FREE SUBSCRIPTION TO **The Source**℠

Complete Apple Modem $129

Single-Slot 300 Baud Direct-Connect Modem for Apple II, II+, IIe and Franklin computers





FEATURES OF THE NETWORKER™

1. DIRECT CONNECT – No acoustic coupling needed. Two modular telephone jacks – one for phone – one for line
2. SINGLE CHIP MODEM for greater reliability
3. ON BOARD FIRMWARE contains a terminal program
4. ON BOARD SERIAL INTERFACE – no extra cards to buy. Software selectable data format: 7 or 8 data bits, one or two stop bits, odd or even parity, full or half duplex
5. 300 BAUD software selectable for 110 baud
6. SWITCH CONTROL for answer originate is next to keyboard
7. CARRIER DETECT LED gives you line status at a glance

ALL THIS PLUS . . .

- COMPLETE with NETWORKER SOFTWARE to give you:
  - Text trapping of entire display into RAM memory
  - Disk storage capability for all trapped text
  - On screen menu and status indicators
- FREE SUBSCRIPTION TO THE SOURCE, the popular dial-up information system
- SOFTWARE COMPATIBILITY – with all common Apple communication software
- COMPATIBLE with both rotary and tone phones
- FCC APPROVED – Made in USA
- ONE YEAR MANUFACTURER'S WARRANTY

NETWORKER™ INCLUDES A COMPLETE PACKAGE

- Networker modem card and control software
- Modular phone line cord
- Networker software on a disk ready to run
- Complete instruction manual

NETMASTER™ COMMUNICATIONS SOFTWARE

For $179 we include with the **NETWORKER** the **NETMASTER** Communications Software for advanced users. **NETMASTER** will let you transfer games, computer graphics, programs, sales reports, documents – in fact any Apple file of any size – to another computer directly from disk to disk without errors, even through noisy phone lines.

JQB ENTERPRISES INC
P.O. Box 269
Ayer MA 01432

For transfering information between computers, **NETMASTER**'s superb error checking and high speed are an unbeatable combination. With a **NETMASTER** on each end, you can transfer information three to five times faster than other communications packages like Visiterm™ or ASCII™ Express. Error free.

Your best buy in modem history. The **Networker,**™ a plug-in single-slot direct connect modem for the Apple II family of computers. Send electronic mail to a friend or business associate, use your school's computer, access hundreds of computer bulletin boards or thousands of data bases for up-to-the-minute news, sports, weather, airline, and stock information.

There's absolutely nothing else to buy. You get the modem board, communication software, and a valuable subscription to America's premier information service, **THE SOURCE**™. For $129 it's an unbeatable value.

This is the modem that does it all, and does it for less. The Apple Communications Card is on board, so no other interface is needed. It's 300 baud, the most commonly used modem speed. And it comes complete with **NETWORKER** Communications Software on an Apple-compatible disk, giving you features *no* modem offers.

Like the ability to lock on-screen messages into your Apple's **RAM,** and then move the information onto a disk for easy reference and review. A terminal program that turns your computer into a communications command center with on-screen "help" menus, continuous updates of memory usage, carrier presence, and communication status.

But **NETMASTER**'s not stuffy. It will talk to those other communications packages, but they don't work as fast and they don't check errors like **NETMASTER.** And **NETMASTER** doesn't only work with the **NETWORKER** modem. Even if you already have another modem for your Apple, **NETMASTER** is an outstanding value in communications software, so we sell **NETMASTER** by itself for $79. **NETMASTER** requires 48k of RAM, one disk drive, and the **NETWORKER** or another modem.

WE EVEN GIVE YOU SOMEONE TO TALK TO!

Your purchase of the **NETWORKER** with or without **NETMASTER** comes complete with a membership to **THE SOURCE,** with its normal registration fee fully waived. **THE SOURCE** will put a world of electronic information and communication services at your fingertips – instantly. Electronic mail and computer conferencing. Current news and sports. Valuable business and financial information. Travel services. A wealth of information about personal computing. Even games. All fully compatible with your equipment, and ready to use at once.

To Order Call Toll Free

800-824-7888 Continental US
800-824-7919 Alaska and Hawaii
or anywhere in the world
916-929-9091
Ask for operator #592

MAIL ORDERS

PLEASE WRITE NUMBER OF ITEMS IN BOX

☐ NETWORKER $129 ☐ NETMASTER $79
☐ NETWORKER/NETMASTER COMBO $179

NAME _____
ADDRESS _____
CITY _____ STATE _____ ZIP _____

Mass. residents add 5% sales tax
Total Enclosed _____

VISA master charge C.O.D.
C.O.D. ORDERS ADD $3.00

☐ MASTERCARD ☐ VISA ☐ CHECK ☐ C.O.D.
CARD NUMBER _____ EXPIRES _____

SIGNATURE _____
(Credit Card orders must be signed)

Send Orders and Make Checks Payable to

JQB Enterprises Incorporated

P.O. BOX 269, AYER MASSACHUSETTS 01432
All Prices Quoted are for Prepaid Orders — Prices Subject to Change Without Notice

Advertiser's Index

Coming in November —

☐ **Building a High-Performance Low-Cost 68000 Microcomputer System**
by Henning Spruth
Build this 68000 system that interfaces to your microcomputer to make use of the I/O capabilities you already own. This multi-part project includes the schematic and part lists, a 68000 resident monitor and a 68000 cross assembler.

☐ **Solid Shape Drawing**
by Richard Rylander
Part 1 of a multi-part series that will expand your understanding of graphics as well as your microcomputer's graphic capabilities. It includes a set of programs and routines that assist in generating more complex shapes complete with realistic shading.

☐ **Fast Plotting of Lines and Functions on the Commodore 64**
by Loren W. Wright
Part 2 of this series provides routines to draw straight lines between points and other simple functions - using fast assembly language subroutines callable from BASIC.

. . . and many other features for the Serious Computerist!

MicroDisks

The Best Programs From MICRO

Ready-to-Run

MD-1 Master Disk Directory
Apple *Charles Hill ..... Dec 1983/Feb 1984*
A utility that collects all of your disk directories onto a single disk which may be sorted and printed. Equivalent to some commercial packages that sell for over $100.00!

MD-2 DOES-IT Monitor
C64 *Michael J. Keryan ..... Jan — May 1984*
An integrated set of assembly language routines that expand the capabilities of your Commodore 64. Included are help screens for the DOS Wedge, a timer/alarm function, many useful functions that may be called directly from the keyboard *via* the RESTORE key — without disturbing the currently running program, a debugging monitor, procedures to 'hide' machine language and BASIC program under ROM, and much more. A must for all serious Commodore users. (NOTE: A complete package including some new features plus complete documentation is now available for $29.95. See the ad elsewhere in this issue)

MD-4 Graphic Printer Dump
C64 *Michael J. Keryan ..... July — Sept 1984*
A printer utility that interfaces with five of the main graphic packages to provide extraordinary printer output on a variety of printers. The prints are full size and may even be in color — with a regular printer! Get the most out of your graphic efforts.

MD-5 CMPRSS: Improved Compression Program
Apple *Ian. R. Humphreys ..... July 1984*
Compress your Applesoft code by:
- Concatenating statements;
- Removing text of REM statements;
- Removing LETs;
- Removing variable names from NEXT statements; and,
- Truncating variable names.

CMPRSS allows you to freely comment your programs, without paying the penalty of running out of variable space.

MD-6 Least Squares Curve Fitting and
Apple Time Series Forecasting
Brian Flynn .... March 1984/September 1984
MD-7
Atari
MD-8
C64
Two sophisticated data analysis techniques are presented in easy-to-use programs. The Least Squares Curve Fitter accepts a series of data and calculates the t-statistic, F-statistic, standard error of the estimate and the Durbin-Watson statistic. The Time Series Forecasting program accepts periodic data (weekly, monthly, annually, ...) and provides five different techniques for forecasting the future trends from the observed data. These two programs provide the basis for stock analysis, scientific data reduction and more.

MD-9 HILISTER
Apple *J. Morris Prosser ..... July/August 1984*
A machine language program that can list to the screen an Applesoft program, a block of disassembled memory, a disk catalog, a memory dump in hex and/or ASCII, or almost anything else. It then supports moving to the beginning or end of the information, 'paging' through one screenful-at-a-time, moving forward or backward one line-at-a time, and optionally highlighting any individual line by inverting it.

MD-10 Fast Bit Map Plotting
C64 *Loren W. Wright ..... Oct/November 1984*
A collection of assembly level subroutines to perform high-speed plotting of points, lines and other functions. These may be readily called from BASIC, or may be used as the basis for custom graphic packages.
(This MicroDisk will be available in November)

MICRO, P.O. Box 6502, Chelmsford, MA 01824
617/256-3649

Mass. Residents add 5% sales tax.

Ordering Instructions:
To avoid errors, please order by Number, Name and Microcomputer.
MicroDisks are $15.00 each, including shipping. MicroDisks are only $10.00 each to Micro Subscribers. You must include the subscription number from your MICRO label when ordering to qualify for this special price.

MD No.	Program Name	Microcomputer

Name _____

Address _____

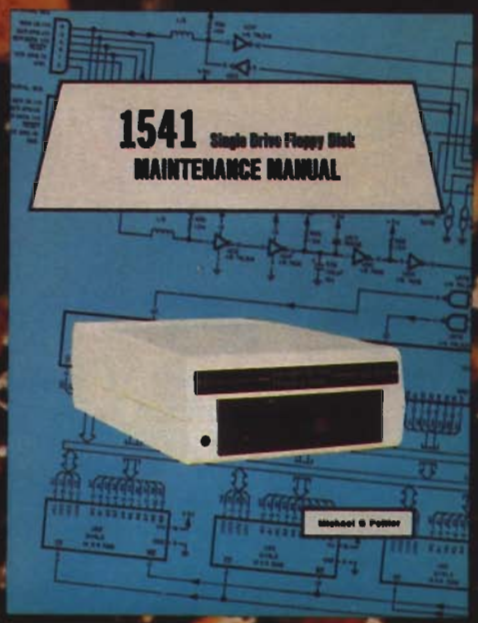
City _____ State _____ Zip _____

☐ VISA ☐ MasterCard ☐ Check

Acct No. _____ Expires _____

Enter Subscriber Number from MICRO Label for Discount

MAINTAIN YOUR 1541 DISK DRIVE



Using the 1541 SINGLE DRIVE FLOPPY DISK MAINTENANCE MANUAL By Michael G. Peltier (198 pages, 118 illus., $29.95)

Or use the 1541 MAINTENANCE GUIDE By Michael G. Peltier (64 pages, 51 illus., $9.95)

Keep your disk drive on track with the ARD-101 (Alignment Reference Disk for 1541 disk drives, $15.95)

For dealer inquiries contact authorized PI distributor:

Computer Marketing SERVICES INC.

26 Springdale Road
Cherry Hill, New Jersey 08003
(800) 222-0585

For distributor and other inquiries contact:

PI phone [316] 945-9266

PELTIER INDUSTRIES

INCORPORATED / 735 N. Doris / Wichita, KS 67212