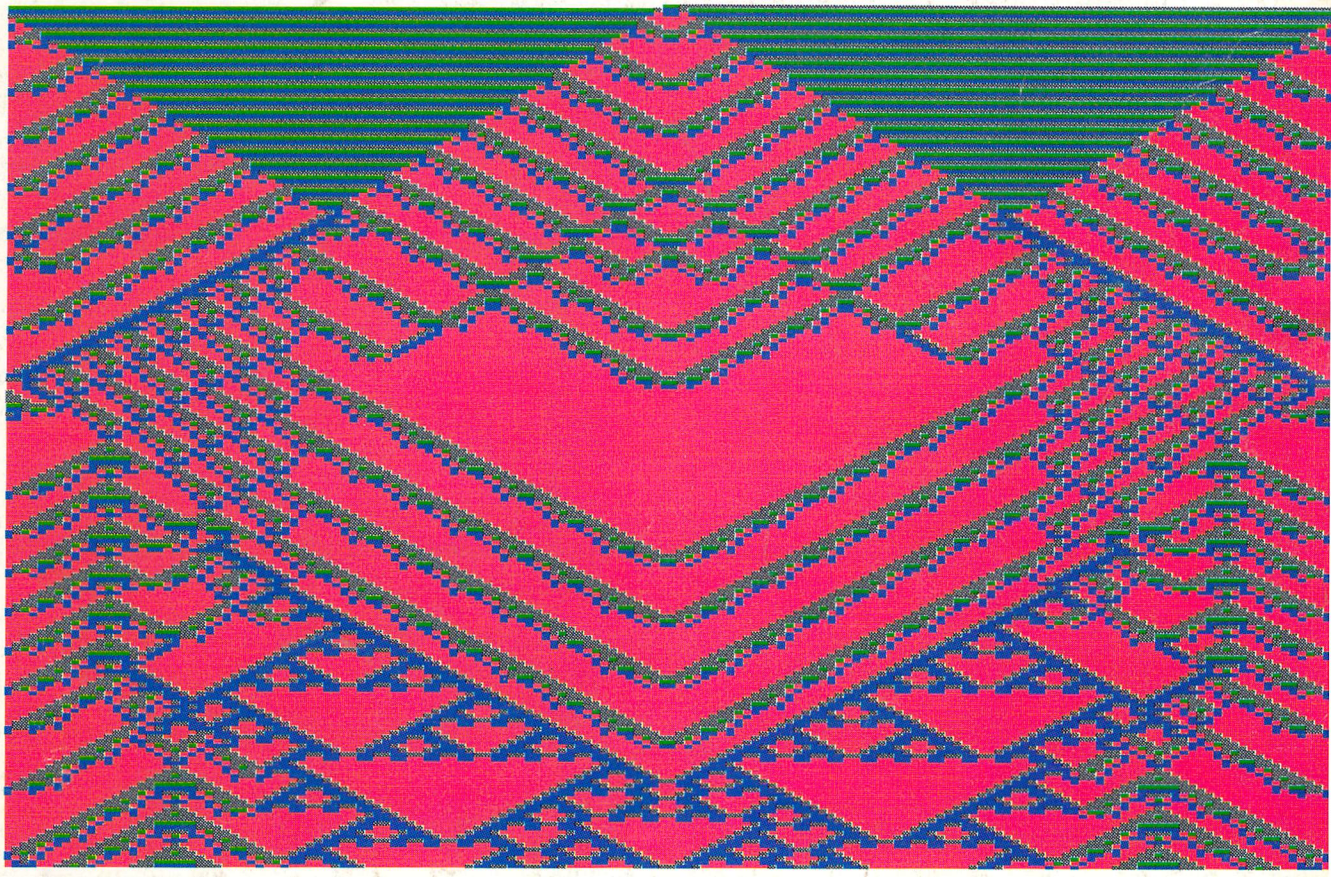
THE MAGAZINE FOR COMMODORE PROGRAMMERS
MAY 1988: VOLUME 8, ISSUE 6

Canada $4.25
USA $3.50

Transactor

- Micro-Lisp 2.5: a Lisp interpreter for the Commodore 64!
- Report on CoNIX - a Unix-style enhancement for CP/M 3+
- Great Assignment: Add expression evaluation to BASIC
- Understanding BRK - get debugging duty from the zero byte
- Fast mnemonic-to-opcode conversion
- An update to "Shiloh's Raid"
- Stabilizing TI$ for long-term timing applications
- Memory swapping routines for screens and programs



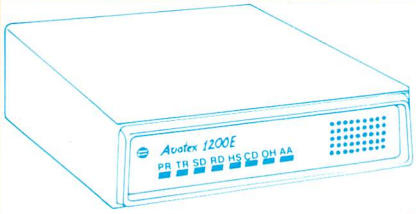
Cellular Automata - A world of mathematical patterns
page 16

0 05
0 29074 13131

diverse
LOGIC

We've made talk cheap

By making available the new Avatex 1200e and SupraModem 2400 we will introduce you to high speed communications without a high speed price.



NEW!
Avatex™ 1200e

only $139.95

- 300/1200 baud operation
- Call progress monitoring
- 2 year warranty
- TOTAL Hayes compatibility
- Includes free hour and package on-line time and introductory subscription to **CompuServe** (a $29.99 value)

SupraModem™ 2400

only $249.95

- 300/1200/2400 baud operation
- 1 year warranty
- Non-volatile memory stores user configuration and last dialed number even when turned off

ADDITIONAL FEATURES

- Hayes compatible
- Built-in speaker
- Bell 103/212A, CCITT V.22 and CCITT V.21 compatible
- Programmable speaker volume
- Full/half duplex operation
- CSA/FCC approved
- Full "S" register support
- Pulse or tone dialing
- Compact case
- All status LED's
- 100% compatible with industry standard "AT" command set
- External power supply

Free terminal software with each modem purchase.

RS-232 CABLES

- Base price of $15.00 plus $1.00 per foot
- Specify computer connector gender type

Onmitronix Deluxe RS-232 Interface

- For use with Commodore 64, 128, Vic-20, SX 64 and Plus 4 computers
- Supports pins 2 through 8, 12, 20, 22
- Includes 3-foot cable
- Compatible with all standard RS 232 equipment
- Recommended by Commodore and Avatex
- For use with both DTE and DCE equipment

only $49.95

($39.95 Can./$32.95 U.S. with purchase of modem)

PLEASE MAKE SURE ALL INFORMATION IS INCLUDED IN YOUR ORDER.

FREE TERMINAL SOFTWARE WITH EACH MODEM PURCHASE.
PLEASE INDICATE COMPUTER TYPE.

☐ AMIGA ☐ Commodore 64 ☐ Atari ST
☐ Commodore 128 ☐ IBM PC ☐ MacIntosh

QTY.		CANADIAN	U.S.	AMOUNT
____	Avatex 1200e(s)	@ $139.95	@ $114.95 = $	
____	Supra 2400(s)	@ $249.95	@ $199.95 = $	
____	RS-232 Interface(s)	@ $ 49.95	@ $ 39.95 = $	
Subtract $10.00 for each interface purchased with modem				$
____	RS-232 Cables ($15.00 + $1.00 per ft. CAN.) ($12.00 + $ .80 per ft U.S.) Specify male or female for computer end.	@ $ 1.00	@ $ .80 = $	
			Sub-Total $	
			7% Ont. Sales Tax $	
			Total $	

Name: _____
Address: _____
City: _____
Prov.: _____ Postal Code: _____
Phone No.: _____

PAYMENT CAN BE MADE

Money Order ☐ Cheque ☐ VISA ☐ M/C ☐
Card # _____ Expiry Date _____
Cardholder's Name: _____
Signature: _____

All products shipped from inventory within 24 hours. Shipping is FREE.

SEND TO

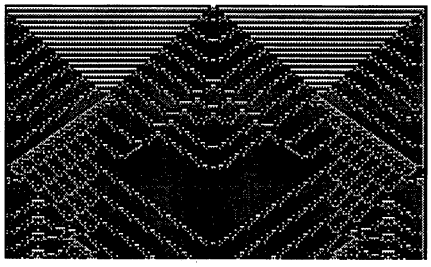
diverse
LOGIC

127 Hillsmount Cres.
London, Ontario
Canada N6K 1V6

Tel.: (519) 657-7841
BBS: (519) 472-5354
(Punter Net node #15)

Transactor Volume 8 Issue 6

About The Cover: This graphic was produced with Ian Adam's "Cellular Automata" program (see article on page 16). This particular pattern was created with the code 2001313120 22. Ian Adam comments, "Very powerful patterns are created from all incarnations of this code. The initial blue-green background gives way to solid red, overlaid by black diagonals and blue branching structures." The colour separations for the cover were done on an Amiga 2000, and typeset with the same equipment used for the rest of the magazine.



Transactor
The Magazine for Commodore Programmers

Publisher
Richard Evers

Editors
Nick Sullivan
Chris Zamara

Editorial Assistant/Advertising
Moya Drummond

Customer Service
Renanne Turner

Contributing Writers

Ian Adam
David Archibald
Jack Bedard
Paul Blair
Glen Bodie
Bill Brier
Anthony Bryant
Jim Butterfield
Dale Castello
Tom Collopy
Richard Curcio
Don Currie
Robert Davis
Elizabeth Deal
Frank DiGioia
Chris Dunn
Paul Durrant
Michael Erskine
Jack Farrah
Mark Farris
Jim Frost
Miklos Garamszeghy
Eric Germain
David Godshall
Michael Graham
Eric Giguère
Thomas Gurley
Patrick Hawley
Adam Herst
Thomas Henry
John Houghton
Robert Huehn
Tom Hughes
David Jankowski
Clifton Karnes
Lorne Klassen
Jesse Knight
Gregory Knox
David Lathrop
James Lisowski
Richard Lucas
Scott Maclean
Steve McCrystal
Chris Miller
Keath Milligan
Terry Montgomery
Ralph Morril
D.J. Morriss
Michael Mossman
Bryce Nesbitt
Gerald Neufeld
Noel Nyman
Helen Olsen
Matthew Palcic
Richard Perrit
Steve Punter
Raymond Quirling
Doug Resenbeck
Tony Romer
Herb Rose
E.J. Schmahl
David Shiloh
Darren Spruyt
Aubrey Stanley
David Stidolph
Richard Stringer
Anton Treuenfels
Audrys Vilkas
Nicholas Vrtis
Jack Weaver
Geoffrey Welch
Evan Williams

Production
In-house with Amiga and Professional Page
Final Typesetting by Vellum Print & Graphic Services Inc.

Printing
Printed in Canada by
MacLean Hunter Printing

Subscription and Order Information

Address all orders, queries and other correspondence to:

Transactor
85 West Wilmot Street, Unit 10
Richmond Hill, Ontario, Canada
L4B 1K7

Or, in the U.S.:

Transactor
P.O. Box 338, Station C
Buffalo, NY
14209

In the U.K., Europe, South Ireland and Scandinavia:

Transactor (UK) Limited
2 Langdale Grove
Bingham, Nottingham
England NG13 8SR
Phone 011 44 949 39380

Note that all mail sent to our Buffalo P.O. box is forwarded to our Richmond Hill office; for fastest delivery, mail directly to the Richmond Hill address above.

For VISA and Mastercard phone orders of subscriptions or other Transactor products, call our new

TOLL FREE ORDER LINE (In the U.S. only)
1-800-248-2719 Extension 911

Please note that the toll-free number is for orders *only*.

In Canada, call our office directly at **(416) 764-5273** on weekdays 9-5 (EST). For queries about existing subscriptions or orders, please call on Mondays, Wednesdays and Fridays only. For prices, products and back issues in stock, etc. see order card in the centerfold.

Authors: Write to the above address for a copy of our writer's guide. Submit articles to the above address, preferably on disk along with any associated programs. Just about any format is acceptable, but articles should preferably be straight ASCII text, without control characters or formatting codes. Minimum payment for articles is $40.00 per printed page. Manuscripts should be typewritten (or computer-printed), double spaced, with special characters or formatting clearly marked. Photos should be glossy black and white prints. Illustrations should be on white paper with black ink only. All material accepted becomes the property of Transactor, unless a special agreement is made for the author to retain copyright. Otherwise, all material is copyright by Transactor Publishing Inc

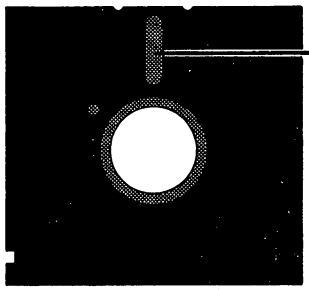
Quantity Orders

In Canada: Ingram Software Ltd., 141 Adesso Drive, Concord, Ontario L4K 2W7. Phone (416) 738-1700

In the U.S.A: IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California 92121. Phone (619) 481-5928. Ask for Dave Bruescher.

The opinions expressed in contributed articles are not necessarily those of Transactor. Although accuracy is a major objective, Transactor cannot assume liability for errors in articles or programs. Transactor Publishing Inc. is a wholly owned subsidiary of Mantra Communications Inc., 95 Lawrence Ave., Richmond Hill, Ont.. L4C 1Z2.

Transactor is published bi-monthly by Transactor Publishing Inc., 85 West Wilmot Street Unit 10, Richmond Hill, Ont. L4B 1K7. Canadian Second Class mail registration number **6342**. USPS **725-050**, Second Class postage paid at Buffalo, NY for U.S. subscribers. U.S. Address: 127 Reed St., Buffalo, NY 14212. U.S. Postmasters: send address changes to Transactor, P.O. Box 338, Station C, Buffalo, NY, 14209. **ISSN# 0827-2530**.



Start Address

Signs of Maturity

You may have noticed that this issue is a little thinner than usual - 64 pages instead of the customary 80. This feature is available for this issue only, and will not be repeated again in the near future. You may have also noticed that this issue is a few weeks late. This relatively minor glitch in the course of the magazine's seven year history comes as a result of some not-so-minor internal changes.

Every *Transactor* since Volume 4 Issue 1 almost five years ago has been produced using professional typesetting equipment, with all the typesetting work done by Karl Hildon, founder and long-time Editor-In-Chief of *Transactor*. This issue is the first created without the benefit of Karl's assistance; he has recently left the company to pursue other interests. Events leading up to Karl's resignation happened quickly, and all of a sudden we had a late issue on our hands, with no apparent way to produce it.

The fact that you are reading this now gives away the happy ending to this tale, but the story is still worth telling, because our leap from editing to typesetting and final production is a success story that says a lot about the maturity of present microcomputer technology. The ease with which we made this leap would have been unheard of just a few years ago, and it's worth looking at what made it possible in the current technological climate.

At the time we were faced with creating the magazine, we knew nothing about "Desktop Publishing" programs or laser printers, and were only vaguely aware of a standard protocol for output devices called PostScript®. Within 24 hours of that terror-filled moment, we were creating pages as you see them here (almost), using our friendly 3 1/2 megabyte Amiga 2000 at the office. Not only can we set text as well as before, but we are able to do things that were previously difficult or impossible, or things that our printer had to do for us.

When we learned a bit about PostScript, we started realizing how little we would have to give up in the quality of the magazine. PostScript is a language that Desktop Publishing programs can use to speak to output devices, like laser printers and phototypesetters. The initial appeal of PostScript to us was that it was a standard, so we could get any program that spoke PostScript and plug it in to any output device that understood it. To our delight, we found PostScript to be powerful and high-level enough that many complicated images can be created quite easily directly in the language.

After learning a bit about PostScript and buying a laser printer came the big step: software. Our choices were limited to PostScript-compatible programs available on the Amiga, which left one obvious choice: Gold Disk's Professional Page, just released. Fortunately, that wasn't a severe limitation, because the quality of the copy it produces is close enough to a professional system that it isn't an issue (although a 300 DPI laser printer doesn't do justice to its capabilities). Just as important, the program is extremely easy to learn and use, which was a major factor in us getting up and running so soon.

The only hurdle left between our quick'n'dirty homemade type-shop and a professional system was the laser printer's less-than-perfect resolution. This hurdle was jumped quite easily by making an arrangement with a local shop that had a PostScript-driven phototypesetting machine: we supply PostScript files generated by Professional Page, and they pump them through the machine and deliver camera-ready film for a reasonable charge per page.

So what does all of this say about the maturity of the technology? Well, we (the user) started without knowing anything, except how to work an Amiga and create the contents of a magazine. We bought a printer and just plugged it in to our Commodore equipment with the enclosed cable; we bought a program from another manufacturer and ran it. Everything worked. Not by accident, but because developers of computers, software, peripherals, languages and standards are working together more tightly to *make* these things work.

We hope you enjoy this special extra-lite issue, with new home-made production goodness. You can expect future issues to grow to regular size, and expect the production to get slicker and the number of diagrams and photos to increase as we get more comfortable with the new system. Ironically, Karl's traditional sign-off is more appropriate now than ever: "There's nothing as constant as change."

Using "VERIFIZER"

The Transactor's Foolproof Program Entry Method

VERIFIZER should be run before typing in any long program from the pages of The Transactor. It will let you check your work line by line as you enter the program, and catch frustrating typing errors. The VERIFIZER concept works by displaying a two-letter code for each program line which you can check against the corresponding code in the program listing.

There are five versions of VERIFIZER here; one for PET/CBMs, VIC or C64, Plus 4, C128, and B128. Enter the applicable program and RUN it. If you get a data or checksum error, re-check the program and keep trying until all goes well. You should SAVE the program, since you'll want to use it every time you enter one of our programs. Once you've RUN the loader, remember to enter NEW to purge BASIC text space. Then turn VERIFIZER on with:

SYS 634 to enable the PET/CBM version (off: SYS 637)
SYS 828 to enable the C64/VIC version (off: SYS 831)
SYS 3072,1 to enable the C128 version (off: SYS 3072,0)

Once VERIFIZER is on, every time you press RETURN on a program line a two-letter report code will appear on the top left of the screen in reverse field. Note that these letters are in uppercase and will appear as graphics characters unless you are in upper/lowercase mode (press shift/Commodore on C64/VIC).

Note: If a report code is missing (or "--") it means we've edited that line at the last minute which changes the report code. However, this will only happen occasionally and usually only on REM statements.

With VERIFIZER on, just enter the program from the magazine normally, checking each report code after you press RETURN on a line. If the code doesn't match up with the letters printed in the box beside the listing, you can re-check and correct the line, then try again. If you wish, you can LIST a range of lines, then type RETURN over each in succession while checking the report codes as they appear. Once the program has been properly entered, be sure to turn VERIFIZER off with the SYS indicated above before you do anything else.

VERIFIZER will catch transposition errors like POKE 52381,0 instead of POKE 53281,0. However, VERIFIZER uses a "weighted checksum technique" that can be fooled if you try hard enough; transposing two sets of 4 characters will produce the same report code but this should never happen short of deliberately (verifizer could have been designed to be more complex, but the report codes would need to be longer, and using it would be more trouble than checking code manually). VERIFIZER ignores spaces, so you may add or omit spaces from the listed program at will (providing you don't split up keywords!). Standard keyword abbreviations (like nE instead of next) will not affect the VERIFIZER report code.

Technical info: VIC/C64 VERIFIZER resides in the cassette buffer, so if you're using a datasette be aware that tape operations can be dangerous to its health. As far as compatibility with other utilities goes, VERIFIZER shouldn't cause any problems since it works through the BASIC warm-start link and jumps to the original destination of the link after it's finished. When disabled, it restores the link to its original contents.

PET/CBM VERIFIZER (BASIC 2.0 or 4.0)

```
CI 10 rem* data loader for "verifizer 4.0" *
CF 15 rem pet version
LI 20 cs=0
HC 30 for i=634 to 754:read a:poke i,a
DH 40 cs=cs+a:next i
GK 50 :
OG 60 if cs<>15580 then print"***** data error *****": end
JO 70 rem sys 634
AF 80 end
IN 100 :
ON 1000 data 76, 138,   2, 120, 173, 163,   2, 133, 144
IB 1010 data 173, 164,   2, 133, 145,  88,  96, 120, 165
CK 1020 data 145, 201,   2, 240,  16, 141, 164,   2, 165
EB 1030 data 144, 141, 163,   2, 169, 165, 133, 144, 169
HE 1040 data   2, 133, 145,  88,  96,  85, 228, 165, 217
OI 1050 data 201,  13, 208,  62, 165, 167, 208,  58, 173
JB 1060 data 254,   1, 133, 251, 162,   0, 134, 253, 189
PA 1070 data   0,   2, 168, 201,  32, 240,  15, 230, 253
HE 1080 data 165, 253,  41,   3, 133, 254,  32, 236,   2
EL 1090 data 198, 254,  16, 249, 232, 152, 208, 229, 165
LA 1100 data 251,  41,  15,  24, 105, 193, 141,   0, 128
KI 1110 data 165, 251,  74,  74,  74,  74,  24, 105, 193
EB 1120 data 141,   1, 128, 108, 163,   2, 152,  24, 101
DM 1130 data 251, 133, 251,  96
```

VIC/C64 VERIFIZER

```
KE 10 rem* data loader for "verifizer" *
JF 15 rem vic/64 version
LI 20 cs=0
BE 30 for i=828 to 958:read a:poke i,a
DH 40 cs=cs+a:next i
GK 50 :
FH 60 if cs<>14755 then print"***** data error *****": end
KP 70 rem sys 828
AF 80 end
IN 100 :
EC 1000 data 76,  74,   3, 165, 251, 141,   2,   3, 165
EP 1010 data 252, 141,   3,   3,  96, 173,   3,   3, 201
OC 1020 data   3, 240,  17, 133, 252, 173,   2,   3, 133
MN 1030 data 251, 169,  99, 141,   2,   3, 169,   3, 141
MG 1040 data   3,   3,  96, 173, 254,   1, 133,  89, 162
DM 1050 data   0, 160,   0, 189,   0,   2, 240,  22, 201
CA 1060 data  32, 240,  15, 133,  91, 200, 152,  41,   3
NG 1070 data 133,  90,  32, 183,   3, 198,  90,  16, 249
OK 1080 data 232, 208, 229,  56,  32, 240, 255, 169,  19
AN 1090 data  32, 210, 255, 169,  18,  32, 210, 255, 165
GH 1100 data  89,  41,  15,  24, 105,  97,  32, 210, 255
JC 1110 data 165,  89,  74,  74,  74,  74,  24, 105,  97
EP 1120 data  32, 210, 255, 169, 146,  32, 210, 255,  24
MH 1130 data  32, 240, 255, 108, 251,   0, 165,  91,  24
BH 1140 data 101,  89, 133,  89,  96
```

VIC/64 Double Verifizer — Steven Walley, Sunnymead, CA

When using 'VERIFIZER' with some TVs, the upper left corner of the screen is cut off, hiding the verifizer-displayed codes. DOUBLE VERI-

FIZER solves that problem by showing the two-letter verifizer code on both the first and second row of the TV screen. Just run the below program once the regular Verifizer is activated.

```
KM 100 for ad=679 to 720:read da:poke ad,da:next ad
BC 110 sys 679: print: print
DI 120 print"double verifizer activated":new
GD 130 data 120, 169, 180, 141,  20,   3
IN 140 data 169,   2, 141,  21,   3,  88
EN 150 data 96,  162,   0, 189,   0, 216
KG 160 data 157,  40, 216, 232, 224,   2
KO 170 data 208, 245, 162,   0, 189,   0
FM 180 data   4, 157,  40,   4, 232, 224
LP 190 data   2, 208, 245,  76,  49, 234
```

VERIFIZER For Tape Users **Tom Potts, Rowley, MA**

The following modifications to the Verifizer loader will allow VIC and 64 owners with Datasettes to use the Verifizer directly (without the loader). After running the new loader, you'll have a special copy of the Verifizer program which can be loaded from tape without disrupting the program in memory. Make the following additions and changes to the VIC/64 VERIFIZER loader:

```
NB  30   for i=850 to 980: read a: poke i,a
AL  60   if cs<>14821 then print"*****data error*****": end
IB  70   rem sys850 on, sys853 off
--  80   delete line
--  100  delete line
OC 1000 data  76,  96,   3, 165, 251, 141,   2,   3, 165
MO 1030 data 251, 169, 121, 141,   2,   3, 169,   3, 141
EG 1070 data 133,  90,  32, 205,   3, 198,  90,  16, 249
BD 2000 a$="verifizer.sys850[space]"
KH 2010 for i=850 to 980
GL 2020 a$=a$+chr$(peek(i)): next
DC 2030 open 1,1,1,a$: close 1
IP 2040 end
```

Now RUN, pressing PLAY and RECORD when prompted to do so (use a rewound tape for easy future access). To use the special Verifizer that has just been created, first load the program you wish to verify or review into your computer from either tape or disk. Next insert the tape created above and be sure that it is rewound. Then enter in direct mode: OPEN1:CLOSE1. Press PLAY when prompted by the computer, and wait while the special Verifizer loads into the tape buffer. Once loaded, the screen will show FOUND VERIFIZER.SYS850. To activate, enter SYS 850 (not the 828 as in the original program). To de-activate, use SYS 853.

If you are going to use tape to SAVE a program, you must de-activate (SYS 853) since VERIFIZER moves some of the internal pointers used during a SAVE operation. Attempting a SAVE without turning off VERIFIZER first will usually result in a crash. If you wish to use VERIFIZER again after using the tape, you'll have to reload it with the OPEN1:CLOSE1 commands.

C128 VERIFIZER (40 column mode)

```
PK 1000 rem * data loader for "verifizer c128"
AK 1010 rem * commodore c128 version
JK 1020 rem * use in 40 column mode only!
NH 1030 cs=0
OG 1040 for j=3072 to 3214: read x: poke j,x: ch=ch+x: next
JP 1050 if ch<>17860 then print "checksum error": stop
MP 1060 print "sys 3072,1: rem to enable"
AG 1070 print "sys 3072,0: rem to disable"
ID 1080 end
GF 1090 data 208,  11, 165, 253, 141,   2,   3, 165
MG 1100 data 254, 141,   3,   3,  96, 173,   3,   3
HE 1110 data 201,  12, 240,  17, 133, 254, 173,   2
LM 1120 data   3, 133, 253, 169,  38, 141,   2,   3
JA 1130 data 169,  12, 141,   3,   3,  96, 165,  22
EI 1140 data 133, 250, 162,   0, 160,   0, 189,   0
KJ 1150 data   2, 201,  48, 144,   7, 201,  58, 176
DH 1160 data   3, 232, 208, 242, 189,   0,   2, 240
JM 1170 data  22, 201,  32, 240,  15, 133, 252, 200
KG 1180 data 152,  41,   3, 133, 251,  32, 135,  12
EF 1190 data 198, 251,  16, 249, 232, 208, 229,  56
CG 1200 data  32, 240, 255, 169,  19,  32, 210, 255
EC 1210 data 169,  18,  32, 210, 255, 165, 250,  41
AC 1220 data  15,  24, 105, 193,  32, 210, 255, 165
JA 1230 data 250,  74,  74,  74,  74,  24, 105, 193
CC 1240 data  32, 210, 255, 169, 146,  32, 210, 255
BO 1250 data  24,  32, 240, 255, 108, 253,   0, 165
PD 1260 data 252,  24, 101, 250, 133, 250,  96
```

Introducing The Standard Transactor Program Generator

If you type in programs from the magazine, you might be able to save yourself some work with the program listed on this page. Since many programs are printed in the form of a BASIC "program generator", which creates a machine language program on disk, we have created a "standard generator" program that contains code common to all program generators. Just type this in once, and save all that typing for every other program generator you enter!

Once the program is typed in (check the verifizer codes as usual when entering it), save it on a disk for future use. Whenever you type in a program generator (for example listings 5 and 6 from the article "Interfacing two Commodore 64s" in this issue), the listing will refer to the standard generator. Load the standard generator you've saved, then type the lines from the listing as shown. The resulting program will include the generator code and be ready to run.

When you run this new generator, it will create a machine language program on disk that can be loaded (load"filename",8,1) and executed with a SYS command. The machine language program is described in the related article, and the generator is just an easy way for you to create it using the standard BASIC editor at your disposal. After the machine language file has been created, the generator is no longer needed. The standard generator, however, should be kept handy for all future Transactor type-in program generators.

The standard generator listed here will appear in every issue from now on (when necessary) as a standard Transactor utility like Verifizer.

```
MG 100 rem  transactor standard program generator
EE 110 n$="filename": rem  name of program
LK 120 nd=000: sa=00000: ch=00000
KO 130 for i=1 to nd: read x
EC 140 ch=ch-x: next
FB 150 if ch then print"data error": stop
DE 160 print"data ok, now creating file"
CM 170 restore
CH 180 open 1,8,1,"0:"+n$
HM 190 hi=int(sa/256): lo=sa-256*hi
NA 200 print#1,chr$(lo)chr$(hi);
KD 210 for i=1 to nd: read x
HE 220 print#1,chr$(x);: next
JL 230 close 1
MP 240 print"prg file '";n$;"' created. . ."
MH 250 print"this generator no longer needed."
IH 260 :
```

b i t s

Got an interesting programming tip, short routine, or an unknown bit of Commodore trivia? Send it in - if we use it in the bits column, we'll credit you in the column and send you a free one-year's subscription to The Transactor

Header With Variables
Ken Garber, Windsor, Ontario

The example in the C128 user guide (page 265) on using HEADER has an error. A syntax error will occur if one attempts to assign the disk ID to a variable name (e.g. I(A$)). This is confirmed by the BASIC 4.0 Reference Manual. However, by manipulating the string variables, the ID can be successfully set from within a program.

As the 'Are you sure?' prompt is suppressed in program mode, the 'escape' prompt in line 40 has been added. Line 30 allows for a short NEW.

```
10 input"diskname";a$: input"disk id";b$
20 c$=a$+","+b$
30 if b$="" then c$=a$
40 input"are you sure ";q$:if q$<>"y" then end
50 header(c$)
60 print ds$
```

C128 64-Mode Autobooter
Aaron Spangler, Everett, Washington

You've heard about all those auto start programs for the C64 that 'save you keystrokes'. Well, this one beats them all - you don't have to hit any keys until your program is loaded. There's one catch - it only works on the 128.

It works like this: First it sets up an autostart program at track 1 sector 0. When the 128 powers up, it checks for the uppercase characters 'CBM'. If it finds it, it executes the machine language following. My machine language program transfers part of its program to $8000 in bank 0, flips to bank 15 and then goes to 64 mode.

When the 64 resets, it checks for 'CBM' then the byte $80 at $8004. If it finds this, it executes the ML pointed to by locations $8000 and $8001. The ML there initializes as the Kernal would until it's ready to go to BASIC, but before it does that, it puts a 'lO"*",8,8' and a shifted RUN/STOP in the keyboard buffer. After BASIC has taken over, it goes to the ready prompt and dumps those characters. The only complication is that when you hit the RESTORE key on the 64, it also checks for the same mask at $8004 and of course it finds it. Then it jumps to the location pointed to by $8002 and $8003. I didn't want the RESTORE key to reset the computer, so I had it point right back into the Kernal. Here's the program that does it all.

```
IB  100 print" 'autoboot 64'  -  by : !the wolverine! "
KA  110 print"this program puts an autoboot program"
GL  120 print"at track: 1 sector: 0.  upon reset, the"
BC  130 print" autoboot program is loaded by the 128,"
ID  140 print" the program sets the 128 into 64 mode,"
CE  150 print"    loads the first program on disk,"
DO  160 print"          and executes it."
HH  170 for x=1 to  91: read a: b=b+a
        : a$=a$+chr$(a): next
LD  180 for x=1 to 164: b$=b$+chr$(0): next
EF  190 if b<>10230 then print"data error": stop
PL  200 data  67,  66,  77,   0,   0,   0,   0,  0
AH  210 data   0, 120,  32, 132, 255, 169,   0, 141
IM  220 data   0, 255, 120, 162, 128, 189,  37,  11
KA  230 data 157, 255, 127, 202, 208, 247, 169,   0
OC  240 data 141,   0, 255,  76,  77, 255,   9, 128
AP  250 data  94, 254, 195, 194, 205,  56,  48, 142
LC  260 data  22, 208,  32, 163, 253,  32,  80, 253
MC  270 data  32,  21, 253,  32,  91, 255,  88, 162
PD  280 data  10, 189,  42, 128, 157, 118,   2, 202
IO  290 data 208, 247, 169,  10, 133, 198, 108,   0
EJ  300 data 160,  76, 207,  34,  42,  34,  44,  56
BM  310 data  44,  49, 131
AD  320 print: print"insert new formated disk &
        press return."
```

```
MP  330 get z$: if z$<>chr$(13) then 250
JO  340 open15,8,15,"b-a:0,1,0"
IJ  350 open5,8,5,"#"
AH  360 print#15,"b-p:5,0"
NM  370 print#5,a$;b$;
GC  380 print#15,"u2:5,0,1,0"
OO  390 close5: close15
```

Ordering Information
For Commodore Hardware Service Manuals
Ted Evers, Richmond Hill, Ontario

Device	Commodore Part Number
C64	#314001-02
8050/8250	#314011-03
1540/1541	#314002-01

2031 (Identical to 1541 long-board except for input circuitry.)
#1540039 sheet numbers 1 and 2 for 2031 schematics.

Message Scroller
Nick Barrowman, St. John's, Newfoundland

Message is an interrupt driven routine to scroll a message across the bottom of the 64's screen. It allows huge messages (which can be composed using *writer*, a short basic program included) to scroll across at any speed you choose. You can set the colour (and change it at any time) by POKEing it into 1023. To set the speed, POKE 780,speed (the lower the faster) and then SYS 49152

```
DM  10 rem ** interrupt message scroller **
HM  20 rem ** by nick barrowman **
CB  30 read a: if a>-1 then ch=ch+a
      : poke 49152+b,a: b=b+1: goto 30
AM  40 if ch<>13439 then print "checksum error!": end
NN  50 print "poke 780,speed"
JM  60 print "poke 1023,colour"
JI  70 print "poke 780,255 to halt"
PK  80 print "sys 49152 to activate"
CL  90 print "note: always poke 780 before the sys"
JO  1000 data 120, 201, 255, 208,  14, 173, 107
PH  1010 data 192, 141,  20,   3, 173, 108, 192
KG  1020 data 141,  21,   3,  88,  96, 141,  61
IJ  1030 data 192, 173,  20,   3, 141, 107, 192
HJ  1040 data 173,  21,   3, 141, 108, 192, 169
JG  1050 data  54, 141,  20,   3, 169, 192, 141
EE  1060 data  21,   3, 169,   0, 133, 251, 169
PB  1070 data 193, 133, 252,  88,  96, 238,  19
GL  1080 data   3, 173,  19,   3, 201,   0, 144
JH  1090 data  42, 169,   0, 141,  19,   3, 160
GB  1100 data   0, 177, 251, 208,  11, 169,   0
CJ  1110 data 133, 251, 169, 193, 133, 252,  76
DL  1120 data 106, 192, 153, 192,   7, 173, 255
EM  1130 data   3, 153, 192, 219, 200, 192,  40
OH  1140 data 208, 227, 230, 251, 208,   2, 230
ON  1150 data 252,  76,   0,   0,  -1
```

```
5 rem the message writer program:
10 a=0: print chr$(147);
20 get a$: print a$;: if a$<>"@" then 20
30 b=peek(1024+a): poke 49408+a,b
   : if b<>0 then a=a+1: goto 30
```

C64 Joystick Port Protection
Gary M. Collins, Bonner Springs, Kansas

The notorious susceptibility of the C-64/C-128 joystick ports to static "zaps" (and resulting severe damage to the computer) can be avoided in many ways. Evan Williams (*Transactor* Vol. 8, Iss. 3, p. 25) solves the problem by killing the static charge before contact with the pins. As he says in his excellent article, however, the pins should *not* be touched at all. One very cheap and simple method is to simply slap a piece of vinyl electrical tape across the unused opening(s).

Another method, equally cheap, involves one of those scrapped joysticks that everyone accumulates. Simply cut the wire flush with the back end of the plug. Dab a bit of 5-minute epoxy on the exposed wire ends, and *presto*! - a dummy plug for the unused port. (I leave a stick plugged into Port 2 all the time.) This makes it easy to move the plug as needed, doesn't gum up the pins as tape might, and eliminates the necessity of opening the computer.

Seikosha Printer Ribbon Reloads
Robert V. Davis, Salina, Kansas

The Seikosha printers, SP-1000VC and SP-180VC, plug directly into the Commodore C-64/128 serial bus and provide reasonable print quality at a very low price. But the replacement ribbons cost about $10 each. Users with a masochistic streak or a desire for economy can reload their ribbon cartridges for about $4.50 each. My first reload took an hour and my hands became very inky. Experience helps. Be very careful opening the plastic cartridge to avoid breaking the pins which hold it together. Follow precisely the directions for installing the reload so the highly compressed ribbon will not spring out of the plastic housing (personal experience).

Reloads for the Tandy DMP-130 printer, packed three to a box, are exact replacements for the Seikosha printers. The Radio Shack part number is 26-1238. Save money!

Gemini 10x Ribbon Refreshing
Dashim Shah, Republic of Singapore

Murray Kalisher's tip on ribbon rejuvenation in BITS (Issue Nov. '87), prompts me to write in with this tip for users of STAR's venerable Gemini 10x series of printers.

These printers, very popular with 64 users, use spool-type ribbons, similar to those used in typewriters, but with one very

important difference. On either end of the ribbon are two eyelets. These eyelets trigger a mechanical lever when either spool is empty, causing the spool to turn the other way, thus allowing the full spool to unwind.

Well, after using my original STAR ribbon for almost two years (that's right, two years!), I discovered that STAR does not supply these ribbons any longer. Since it looked like a typewriter ribbon anyway, I thought it would be available at my normal stationer.

Here is problem number one. It is practically impossible to locate ribbons with eyelets at the ends, in Singapore anyway. Problem number two is that most typewriter spools are not configured with the correct number or spacing of holes to fit the printer.

The solution is rather obvious, if a little messy. Unwind the new ribbon from its original spool and wind it onto the printer's old spools (having removed the old ribbon first of course). As for the eyelets, I simply used a couple of staples, vertically stapled to the new ribbon where the eyelets should have been. It worked beautifully! However, for those who prefer a more elegant approach, you can use a couple of those bulk eyelets that legal firms are so fond of using to hold a bunch of papers together by their corners. Simply punch them into position.

Finally, you can extend the life of your ribbon by just flipping the spool around. This will automatically position the unused half of the ribbon on top, giving you the use of two ribbons for the price of one.

More C-128 Mysteries...
A.J. Saveriano, Sparta, New Jersey

Look at these two sentences and decide what the response of the C-128 would be if you typed them in direct or program mode (type them in exactly as shown):

```
QUIT THAT!
OFFEN,  WORDS ARE  SPELLED STRANGELY
```

Ok... now that you have made your decision, type the first sentence in direct mode and hit RETURN. Do the same for the second sentence. Hmmm... did you get the old faithful SYNTAX ERROR? Surprise! A new error code appears (new for me, at least). UNIMPLEMENTED COMMAND ERROR is your reward. Why? Well, I'm not really sure. However, if you scan the BASIC ROM in Bank 15 from addresses 18089 to 18111 you will see that two keywords appear that we (I) didn't know about. They are QUIT and OFF.

This one-liner will let you see them:

```
10 bank 15:print chr$(14):for i=18089 to 18111
   :print chr$(peek(i));:next
```

Since the error calls itself 'unimplemented', I have to assume that Commodore had plans for these two commands but decided against using them (at least for now). Actually, any command starting with the letters QUIT or OFF will cause the above result. I wonder if there are any others?...

Fast Load Killer
Michael T. Graham, Hopatcong, New Jersey

Some C64 games and applications that read or write to disk don't operate properly when they are loaded using EPYX's FAST LOAD or similar cartridges. The trouble isn't with the load operation itself; the problems occur because FAST LOAD is still enabled when the program is executing. Something in FAST LOAD causes drive errors when non-load disk I/O is attempted. One way around this is to disable the cartridge before loading the program, but this means you'll have to put up with the lethargic pace of the unassisted 1541. There is, however, a better way.

You can load the program using FAST LOAD and then disable the cartridge by inserting the following lines in the program's loader or at the beginning of the program itself:

```
10 sys 58451: sys 65418
20 open 15,8,15,"ui": print#15,"i0": close 15
```

The first line resets the vector tables used by BASIC and the Kernal, effectively disconnecting the cartridge's hooks into the system. The second line resets the disk drive, flushing any special code that was uploaded during the load operation.

This technique also disables 'The Final Cartridge' and should work on other fast-loaders as well. It will also work when no cartridge is installed, making it more universal than calling the cartridge's built in disable routine.

Fade Out (Fade In?)
Geoff Seeley, Bridgewater, Nova Scotia

The following screen dazzler was written for only one reason, to see it work. Be forewarned though, prolonged exposure to this program could freak-out your optic nerve! The program is written completely in machine language and is IRQ driven. Type SYS 49152 to start and to stop the program. (For C64 only).

```
FJ   100 x=49152: ck=0
EA   110 read a: if a=-1 then 130
DL   120 poke x,a: ck=ck+a: x=x+1: goto110
CF   130 if ck<>16891 then print"error in data"
MN   140 print"sys 49152 to start/stop":end
AO   1000 data 173,  20,   3, 201,  49, 240,  13
EE   1010 data 120, 169,  49, 141,  20,   3, 169
DC   1020 data 234, 141,  21,   3,  88,  96, 120
LF   1030 data 169,  33, 141,  20,   3, 169, 192
JH   1040 data 141,  21,   3,  88,  96,  24, 173
BF   1050 data 125, 192, 105, 128, 141, 125, 192
```

```
OE  1060 data 201,   0, 240,   3,  76,  49, 234
IH  1070 data 173, 126, 192, 201,  19, 240,  10
BK  1080 data 238, 126, 192, 173, 126, 192, 168
II  1090 data  76,  71, 192, 169,   0, 141, 126
IL  1100 data 192, 168, 185, 128, 192, 141, 127
PI  1110 data 192, 169,   0, 133, 254, 169, 216
FI  1120 data 133, 255, 160,   0, 173, 127, 192
DK  1130 data 145, 254, 230, 254, 208, 247, 230
HN  1140 data 255, 169, 220, 197, 255, 240,   3
FK  1150 data  76,  86, 192, 169,   0, 141,  32
FM  1160 data 208, 141,  33, 208, 173, 127, 192
LL  1170 data 141, 134,   2,  76,  49, 234,   0
PA  1180 data   0,   0,   1,   1,  15,  15,  12
LA  1190 data  12,  11,  11,   0,   0,   0,   0
KL  1200 data  11,  11,  12,  12,  15,  15,   1
CJ  1210 data   1,  -1
```

Disk Light Flasher
Jeff Spangenberg, Zephyr Cove, Nevada

This program flashes the disk light as a strobe on the 1541 and 1571 drives. It works on the C64, or a C128 in 64 mode. The program also monitors the line so that if a disk command is sent the program exits and executes the command.

```
LF  100 rem  flash disk drive LED on 1541
NM  110 ck=0
EG  120 for x=49152 to 49290: read z: ck=ck+z
ML  130 poke x,z: next
OI  140 if ck<>18154 then print"error": end
DI  150 sys 49152
EB  160 :
AD  170 data 160,  62, 169,   8,  32, 177, 255, 169
MI  180 data 111,  32, 147, 255, 169,  77,  32, 168
NH  190 data 255, 169,  45,  32, 168, 255, 169,  87
HD  200 data  32, 168, 255, 152,  32, 168, 255, 169
MO  210 data   5,  32, 168, 255, 169,   1,  32, 168
KK  220 data 255, 185,  77, 192,  32, 168, 255,  32
MF  230 data 174, 255, 136,  16, 205, 169,   8,  32
FE  240 data 177, 255, 169, 111,  32, 147, 255, 169
AE  250 data  85,  32, 168, 255, 169,  51,  32, 168
HA  260 data 255,  32, 174, 255,  96, 120,  41,   0
FO  270 data 141,   0,  24, 169, 254, 170,  32,  38
EA  280 data   5,  32,  38,   5, 202, 224,   1, 208
CG  290 data 248,  32,  38,   5,  32,  38,   5, 232
DK  300 data 224, 255, 208, 248, 173,   0,  24, 240
PK  310 data 232,  88,  96, 138,  72,  73, 255, 168
CM  320 data 169, 248, 141,   0,  28, 202, 208, 248
LM  330 data 169, 240, 141,   0,  28, 136, 208, 248
DL  340 data 104, 170,  96
```

Best-Bit-From-Moscow Department

Restore to Line Number
Oleg Smirnov, Moscow, U.S.S.R.

I was recently making a MONOPOLY game for my 64. There was lots of data that required random access, but keeping it in arrays would have consumed too much memory. Some BASICs have a RESTORE [line number] statement to do the job. So does the 64's BASIC 2.0, according to some smartie at Commodore (see user's guide, page 176, 'RESTORE line number'). Unfortunately, my 64 never read the user's guide and reported a ?SYNTAX ERROR when put face to face with this version of RESTORE statement. To make up for its ignorance, I wrote this short ML program as a substitute:

```
OF  1 rem 64 restore-r - oleg smirnov
DJ  5 s=49152
EF  10 for i=s to s+54: read a: poke i,a: next
OK  20 data 165,  43, 133, 251 165,  44, 133, 252
IC  25 data 160,   2, 177, 251, 197,  63, 208,  21
BI  30 data 200, 177, 251, 197,  64, 208,  14, 165
GG  35 data 251,  24, 105,   4, 133,  65, 165, 252
ND  40 data 105,   0, 133,  66,  96, 160,   0, 177
LF  45 data 251, 133, 254, 200, 177, 251, 133, 252
BA  50 data 165, 254, 133, 251,  76,   8, 192
```

Here is how to use it. Poke the line number to be RESTORED to into locations 63-64 decimal in the standard low byte/high byte format, and then SYS 49152. Example (dl stands for data line number):

```
100 hi=dl/256: lo=dl-int(hi)*256
105 poke 63,lo: poke 64,hi: sys 49152
```

To relocate RESTORE-r to a different address in memory, change line 5 ('s' is the starting address) and line 50. The last two numbers of line 50 are equal to s+8 in the low byte/high byte format. Change these to correspond to your new starting address plus eight.

RESTORE-r uses zero-page locations $FB-FC and $FE (251-252 and 254 decimal). To use other locations, make changes wherever you encounter a 251, 252 or 254 in the data statements. Also, for the sake of shortness, RESTORE-r lacks error trapping. It gets stuck on non-existent line numbers. Use RUN STOP - RESTORE in this case.

And one more thing. With the following changes, RESTORE-r becomes a computed GOTO routine:

```
Line 25: 57 instead of 63
Line 30: 58 instead of 64
Line 35: 56,233,1,133,122 instead of 24,105,4,133,65
Line 40: 233 instead of 105; 123 instead of 66
```

To use it, poke the line number into locations 57-58 decimal (low byte first), and SYS 49152. Control is then transferred to the line specified. Example:

```
100 input "which line number should I GOTO";g
110 hi=g/256: lo=g-int(hi)*256:
120 poke 57,lo: poke 58,hi
130 sys 49152
```

Letters

Oh, For the Good Old Days... Today, I received the last issue of my subscription. I have carefully considered my decision regarding renewal.

Two years ago, A friend of mine lent me his *Transactor*. At that time, I had just about absorbed all the pablum to be had in Compute!'s Gazette which seemed to repeat the same old stuff over and over, always directed to the rank beginner. God! was I happy to find **T** Because it went to the very heart of my computer, telling me all the hidden secrets which I could never have discovered on my own at that stage of my development.

I have eagerly awaited every issue since that first one because I knew there would be a choice piece of information and/or a terrific program I needed. As I once wrote to you, "It seems that every time I need a program to do a certain job, the very next issue of **T** provides exactly what I need!" It was eerie at times...

For the last two or three issues, however, the quality of the magazine print has soared to new heights but the quality of the words (which is all that matters) has dropped to the level of Compute!'s Gazette. **T** is now full of empty words. Who needs reviews? Who needs opinions? I miss the interesting covers and I hate the slick paper. The very reasons for my loving **T** so much are almost all gone.

I wish you well with your new "look" but from now on, I will just buy a copy of Compute!'s Gazette in the grocery store when one interests me. Someone there at **T** should realize that the old magazine was directed to super-intelligent, avid C-64 fans who crave the arcane knowledge found only in the old **T**. Now, you seem to be trying to grasp the beginner's market but, all of them are gone.

It is very much like watching the slow death of a dear friend and with deep regret that I must decide not to renew my subscription.

P.S. I failed to mention I'm NOT interested in Amiga, Apple, 128, or B-Series - only C-64. If you ever become a "special" magazine again, *please* let me know!

Wayne Gurley
Wills Point, Texas, USA

CP/M Saviours: Once again *Transactor* comes to the rescue. I had just purchased my new 1581 drive and could hardly wait to get it up and running, when I discovered that the current CPM+ system wouldn't support it. I played with it in the BASIC 7.0 mode and it was great! But I really couldn't wait to get *dBASE* and *WordStar* over to one of those "little square" disks.

Then Mike Garamszeghy saved me with his CPM+.SYS patch. Now I can PIP to my 1581 all day long. *Thanks Mike!*

Thanks for the articles on CP/M and thanks to writers like Mr. Garamszeghy, Adam Herst, Clifton Karnes and Aubrey Stanley for sharing their knowledge with other Commodore CP/M users. And thanks to *Transactor* for offering a place for this sharing to take place.

That one issue was worth the price of the subscription. It is a shame that Commodore couldn't have supplied the 1581 system with the drive (but I've been a Commodore user long enough to know all about Commodore Customer Support).

P.S. Mike Garamszeghy is fast becoming the CP/M "Jim Butterfield".

Dr. Ken Flippo
Milan, Tennessee, USA

Contrasting letters like the above two keep us walking an editorial tightrope around here. No matter how we fill these pages, we're bound to get both praise and criticism; as long as the former prevails (and our sales don't dip dramatically), we figure we're doing something right. Wayne Gurley's letter, though, is worth a closer look because his perceptions of the magazine's "slow death" have been voiced by one or two others. Compared against the facts, though, they just aren't true.

*The interesting thing about a magazine like the **T** is that people read it, to a large extent, for the parts they don't understand. We've heard from so many readers, "I love your magazine, even though I don't understand any/most/some of it", that it's a standing joke around here. These people hope to learn by eventually grasping more of the magazine's content, so that the parts they don't understand become less with each passing issue. If we've done our jobs well, you should be finding less (in Wayne's words) "hidden secrets which I could never have discovered on my own at that stage of my development." Why is that surprising, if your whole purpose is to increase your stage of development? Ironically, the better we do our job of educating and informing, the more we remove the Transactor magic from the minds of long-time readers, and it is possible that we may even lose some along the way. (Mr. Gurley's reaction of going back to the "pablum to be had in Compute!'s Gazette" is one whose logic escapes us, however.)*

If you take a good look at the recent issues, you'll find just as much technical info as always (if not more), and I'm sure many will testify that much of it is incomprehensible to them at this stage. No Amiga content appears in this magazine any more, and Apple content never did; expect coverage of the C64 and C128 exclusively, with a tiny smattering of PET/ SuperPET/ VIC/ Plus 4/ B-series related material. If we were going after the beginner's market, the previous issue wouldn't have articles on a communications interface for software developers, hacking the POWER C function library, a detailed look at C128 ROM code to investigate a little-known but deadly bug, bank-switching routines for C128 machine code, and other material that would guarantee instant death in that market.

To address Dr. Flippo's letter, we appreciate the support for Mike Garamszeghy's articles. Feedback like this helps us determine what to put into the magazine; your letters do make a difference!

Chess, Anyone? I recently purchased a C-128D based largely on all prior reports that C-64 software runs OK on it, without exception. That's not the case! *Colossus Chess IV*, from Silver House, England, will not run on my C-128D. It works fine, as it always has, on my C-64. I'd appreciate knowing if others with a C-128 or C-128D have had the same problem. I have tried, but am unable to get a response from Commodore in this regard.

P.S. My C-128D works fine, otherwise. It even works OK with *Superkit 1541*, which I was never able to get to work correctly with my old C-64/1541!? *Except*, despite claims to the contrary, *Superkit 1541*, will *not* copy itself.

John R. Menke, Chessoft Ltd.
Mt. Vernon, Illinois, USA

*Other than slight differences which vary among all C64 versions, we were not aware of any compatibility problems with the 128 or 128D. At this stage, we know little about the 128D other than the fact that it is a repackaged C128. Even subtle differences in hardware or software can break a program however, especially one that uses clever copy protection methods, or code that relies on undocumented quirks in the system. (we don't know if **Colossus** is one of these). If anyone knows of a commercial program that runs on the 128 but not the 128D, we would like to hear about it - let's see if there's a problem, and how widespread it is.*

Believe it or not, we have made working copies of SuperKit using *SuperKit itself - on a brand-new 1571. The program is slick and very high-performance, but, like a highly-tuned racing engine, just a bit too finicky for everyday use in the real world. Perhaps that has something to do with why SuperKit publisher Prism Software is now out of business.*

Random Drive Errors Corrected: It has been noted that a 1571 disk drive without ROM updates will produce random '?device not present error' messages. I encountered this error several times while reading a relative file, and upgraded my 1571 ROM to get rid of this bug. Well, this did not get rid of the errors. I asked Commodore in the United States, a software company in the U.S. and a few store owners in the area - my question was still unanswered. I had deduced that the errors were being caused by a *very* low voltage at sporadic times. A C-128 power supply can only handle so much at one time: two disk drives, a modem, a Mach-128 cartridge, a printer, and a printer interface with a 16k print buffer. In my last breath I asked a very knowledgeable 'guru' on a local bulletin board if he had a solution to my problem. He felt that if I changed my printer interface I would solve the problem. So, I changed from a TurboPrint GT with the buffer to a Super Grafix Jr., and it worked.

Here's the reason: the interface that I had was drawing too much amperage from the cassette port, especially with the 16K printer buffer hanging off of it. I was able to narrow the

problem down by turning off the printer and finding that the errors did not occur. If you are having the same problem as I was please check the interface that you are using.

Duane E. Barry
Cambridge, Ontario, Canada

Thanks for the tip!

Where's the RS232 Interface? I've heard that one of your issues had construction directions for an RS232 interface for modem connection. Could you please answer a few questions for me?

1. Will this interface work on an SX64?
2. What issue has this info in it?
3. Will the VIC/C-64 Verifizer work on the SX64? (It doesn't seem to work for me.)

Looking forward to my new subscription.

Isidro G. Nilsson
Marysville, Washington, USA

First, the answers: 1: Yes; 2: Volume 8 Issue 3; 3: Yes.

Now for some explanations...

Other than some differences that vary on different versions of the 64 itself, the SX is 100% software-compatible with the regular 64 and 64C. It is theoretically hardware compatible as well, but since the unit's physical construction is different, it is wise to verify, as you are doing, that a plug-in piece of hardware will fit in the slot on the top of the machine. Most simple boards that plug into the expansion or user ports will - as is the case with our RS232 interface project that appeared in Volume 8 Issue 3

As far as the Verifizer goes, we have used it many times on our SX without a hitch - double-check your program against the listing (you don't have the benefit of the Verifizer to do this, of course!), or better yet, load the Vic/64 verifizer from any Transactor disk. If that's where you got your Verifizer, suspect a bad disk. If Verifizer from a good Transactor disk gives you the same trouble, you've got bad hardware, or you've discovered a problem we don't know about - please let us know!

Transactor Site Licensing Policy - We, the Fundy C64 User's Group are in receipt of the reference letter, clarifying the Transactor copyright policy. However, we see this as a change in your policy and must protest.

In October 1987, we entered into a subscription with your magazine under the clear understanding that *Transactor* software was available for copy but not resale. For your part, you accepted our subscription, and began to supply us with software.

Therefore, this Club and your magazine have entered into a contract, and as with all contracts, unless otherwise agreed in advance, the terms and conditions of that contract remain in effect for its duration.

Accordingly, the Club expects to receive *Transactor* software, as agreed, and will continue to copy but not resell, as agreed.

Our Club consists of members who, having paid a membership fee, are free to copy the software in our library without charge. This Club policy is a factor in determining to which magazines we subscribe. We know of no other magazines adopting this type of policy and feel that it is a serious mistake which will, in future limit the circulation of your product.

In closing, we would ask you to reconsider your policy, since it will influence our future decisions, and, frankly, we like *Transactor.*

Bob Laws
Software Librarian, Fundy C64 User's Group.

Our copyright policy as printed on page one of the magazine was changed recently to prevent exactly the kind of thing your user group is doing, because it is difficult to sell Transactor Disks when anyone can get them for free at their local user group. Our attitude regarding Transactor programs has not changed: use them for whatever you wish; use our routines in your own programs; give them to your friends. What we are trying to prevent is mass-distribution of Transactor Disks, which we put considerable effort into producing, and we would like to avoid having to compete with our own product supplied by others.

The copyright policy in this issue remains the same, but we added a short note about our attitude toward individual copying of our programs. Note that this does not affect our copyright, but just states that we generally don't mind copies of individual programs being given away, but you should get special permission for mass-distribution or for distribution of collections of Transactor programs (like a Transactor Disk). For user groups, our site-licensing arrangement of $3 per copy made is the standard deal.

You're correct in stating that no other magazine has a policy similar to ours, but for the wrong reason: no other magazine has a policy as ***liberal*** *as ours. Ask* Compute! *how they feel about people giving away their programs.*

If you no longer wish to subscribe to the Transactor Disk as a result of our change of policy, then you may cancel your subscription and get a refund for the remaining disks. Alternatively, you can start charging your members a small fee for obtaining our disks without having to order them from us (for

more money) and wait for them to arrive. We believe that this arrangement still benefits your group; that, of course, is for you to decide.

Piracy: the debate continues - This letter is in response to your editorial in the March issue of *Transactor*. There is an old saying, "Be careful what you wish for, you just might get it." For the benefit of the home computer industry as a whole and software game writers in particular, I hope you don't get your wish in the Weaver vs. Doe case.

There are several statements in your editorial which are incorrect as far as US law is concerned. If judgement is found for the plaintiff, the defendant will not be convicted of anything. This is a civil suit.

Secondly, you can most assuredly collect damages from a minor if he has anything. Also, a judgement is good for seven years. By that time your minor is a young adult and you can get that which he had hoped would be his college tuition or business stake too.

Thirdly, judgements are not expensive. US copyright law grants the plaintiff actual damages plus all legal fees if he prevails. Let your lawyer collect his fee and your damages from the defendant.

I hold a copyright to a book and sympathize with Weaver to the extent that Weaver wishes to recover from the pirate himself. My sympathy diminishes rapidly as Weaver tries to pull the parents, the phone company, the fire department and anyone else he thinks may have some money, into his suit.

Parents normally are not held liable for damages caused by their children, only for their own *culpable* negligence. To attempt to extend liability to parents who know little or nothing about computers and have no interest in learning or in constantly monitoring what their child does with his computer can only result in parents refusing to buy computers for their children. This will tend to maximize protection for copyright holders, but it will also minimize the market for games, computers and magazines. I would not want that.

I don't want to start holding Ma Bell responsible for misuse of modems on regular home phone lines at this time, either. She has just barely accepted the idea of permitting such activity instead of confining it to the special computer lines which are more expensive. There goes Q-Link, all the BBS's, and the other services whose make or break point depends on the home computer user.

No, I can't support your position on that. I can support going after the violator with all vigor. Get his equipment, for now, and don't let up until your judgement is paid in full or the time limit has expired.

I can also support a strong request to all responsible owners and users to report violations which come to their attention. Perhaps holders of software copyrights need an organization to monitor and protect their rights as BMI does for the music industry. I wouldn't know how to contact a copyright owner if I did see his work being distributed illegally.

Interestingly, the US copyright law grants certain express rights to the owner of a copy of copyrighted material. These include the right to sell or otherwise dispose of his copy, to make copies for archival purposes, to re-arrange the material to suit his purposes, to study the copyrighted material in depth, and generally, to enjoy all the benefits of ownership except distribution.

Federal court in Louisiana has just ruled that State's "shrink wrap license" law illegal. If you buy the package, you own that copy and the right to use it as you please, period. This comes as no surprise since the "shrink wrap license" was originated as an effort to deny copy owners the rights that they were intended, by Congress, to have.

Congress has been repeatedly lobbied to reduce the rights of the owner of a copy, in the case of tape recorders, video tape decks and, most recently, digital tape recorders. They have steadfastly refused to do so. I don't expect them to change their position just for software.

The use of copy protection schemes obviously is intended to deny the copy owner certain of his rights and is probably an actionable cause. Further, they only harm the legitimate software user. The pirate has ample tools, provided by a segment of the industry solely supported by their use, to effectively negate such schemes. Last year's best seller list is topped by "Print Shop Graphics Library #1" at a quarter of a million copies sold. It has not one whit of protection. GEOS, on the other hand, is given away with every C64 sold. It is of interest mostly to users who want only to plug it in and push the button. It is so loaded with protection that programmers have shown little interest except to prove that they can crack the protection. I ask you, why protect a program you intend to give away? I suggest their interests would be better served by making the shell disk wide open and protecting, if really necessary, the additional programs for it, which they do sell.

I feel that we all, software users and creators, need to begin co-operating to protect each other's rights, not just our own. Working together, we can run the pirates out of the game. Working at cross purposes we only harm each other and cause them to flourish. Nothing so discourages me as seeing a letter from a novice who hasn't mastered BASIC yet, seeking a way to hide his work. If the rest of us hid our work, he would have no chance to learn. Programmers are not born, we all learned most of what we know from others. If Johnnie von Neuman and Grace Hopper had hidden their knowledge, we would all have had to pursue other hobbies or livelihoods. Yet he, somehow, expects an experienced programmer to teach him how he can avoid teaching others. That is the atmosphere that has been created and I abhor it!

Co-operation and a commitment to protecting all parties rights are the only solutions I can see. I certainly don't want to see "big brother" deciding to do for us what we should have done for ourselves.

Russell K. Prater
Parker, Florida, USA

In praise of C - You guys are reading my mind. The March issue's shift towards using C on the 64 and 128 shows a very smart shift of focus on your part. I vote for continued attention to this language because of the language's popularity on other machines (and of the usefulness of familiarity with C when moving up to the Amiga). Since it is so fast, C provides a wonderful world to explore between the two languages you have promoted thus far, BASIC and assembly language. How about some graphics routines in C?

I bought Power C a month ago and am intrigued. If *Transactor* focuses on using this package (with forays into other implementations like Super C by Abacus) the attention will promote the popularity of the language among the Commodore community. The use of some kind of standard is important, as you have proved by promoting PAL as an assembler.

Besides, even though you consistently use the same software, there is always room in *Transactor* for other products. You have proved that by publishing assembly language from MAE, French Silk, the Commodore Assembler and others. It is nice to see alternative implementations, just to know what the other guys are up to. Keep it up. My vote is for continued focus on C.

One more thing. Since moving from Minnesota to Alabama to work at the University of Alabama-Birmingham, I have become heir to four 8032 CBMs, two 8050 disk drives, a 4023 printer and cables to connect them all. The only thing missing is decent software and the manuals. Without the last few years of *Transactor* and the Inner Space Anthology, I wouldn't be able to do anything with them. Do you have any advice as to where to find a word processor and database for these things at a reasonable price? Since the university is a nonprofit organization, donations from readers who have moved on to bigger and better things would probably be tax deductible. I would appreciate it if you would publish this letter along with my address so that interested readers could contact me.

Craig Ede, Art Dept., 101 Honors House, Univ. of Alabama at Birmingham, Alabama, USA 35294

Paperclip and the 65C816 - Re the review of the Turbo Processor for the C64 on page 56 of *Transactor* V8i4: I think I know why Paperclip won't run on a 65C816. As you may know, Paperclip is not DOS copy protected but is dongle protected. This dongle protection seems to consist of two phases. First, when you run the program, the program is decoded using values from the dongle as a key. Secondly, once the program is up and running, there is a loop which checks for the presence of the dongle, apparently during the IRQ (at least, the cursor stops blinking if the dongle is removed, and starts blinking again if you reinsert it).

Alas the decoding phase appears to use undocumented op codes (at least, they are not officially documented in MOS Technology spec sheets although they have been documented in Raeto West's books and in several other places). Of course it is precisely these undocumented codes which the 65816 family uses for its extra instructions. It would be as if you used undocumented 8080 codes which undoubtedly exist in one form or another and then wondered why the resulting code would not run on a Z80. The 65816 does have a 6502 emulation mode but this, I believe, only affects the width of the registers. Thus INX would work differently if the x-register contained $FF depending on whether the X-register were 16 bits or, in 6502 emulation mode, 8 bits. In any case, it would have had to choose between MOS/Commodore, Rockwell and Synertech 6502's, and I believe the trend for both Rockwell and Synertech has been to treat undefined op codes as NOPs in their 65C02's which has the advantage that you can pass these through to a co-processor.

I suppose the moral is that programmers shouldn't be *too* smart with their tricks. If possible, and it isn't always possible, keep with "official" methods. If you don't need blinding speed or decent graphics let your MS-DOS programs use MS-DOS I/O instructions. Assume your 68000 users are going to insert a 68030 board one of these days. Unless you're being paid by the company that built the computer, try to let your users get away with a not altogether 100% compatible clone. And, unless you want a lynch mob outside your door, make sure your 1541 DOS copy protection doesn't only work on 1541s made on alternate Thursdays in June 1984.

Joel M. Rubin
San Francisco, California, USA

Product Info from Readers - I just received the March *Transactor* and read the Letters column. I think I can help a couple of my fellow readers.

Patrick Demets wants a book. Specifically he needs a book which covers cartridge addressing. May I suggest *Easy Interfacing Projects for the Commodore 64* by Downey, Rindsberg and Isherwood. It is available from Prentice-Hall or from Don Rindsberg, The Bit Stop, 5958 S. Shenandoah Road, Mobile, AL., 36608. This book is 200 pages of goodies ranging from the basics to constructing speech synthesizers, IEEE serial and parallel interfaces, a modem, 8-line multiplexed ADC and a ton of other goodies, with the driver programs where needed. It also has a chapter on building an EPROM burner/reader and covers address range selection as well as any book I have seen. The C64 supports cartridges at $8000, $A000, $E000, $DFE0, $DFF0 or any combination thereof by proper manipulation of the EXROM, GAME, ROMH, ROML, I/O_1 and I/O_2 lines and the address register at $0001.

David Kuhn needs a real-time clock for his C128. Everyone who is interested in computers and knows what a soldering iron looks like should write for a catalog from JAMECO Electronics, 1355 Shoreway Road, Belmont, CA 94002. They have all the chips, connectors and supplies that you just can't find anywhere. They do have a $20 minimum order, but I have trouble keeping my orders *down* to that. On page 30 of the current catalog is a real time clock module, a complete 12 or 24 hour clock in a 16 pin DIP with four bit data line access for computers at only $7.95. It is easy to interface through the user port and serves very well if powered from a battery and 7805 regulator. Don't forget to order the specification sheet with it.

I too have a need. Does anyone know where I can get a schematic and ROM program for a print buffer? I would prefer 8-bit Centronics, but Commodore serial would serve.

Russell K. Prater
Parker, Florida, USA

More Clocks - In response to the inquiry "Clock Setting" (V8I5) as a hacker who also likes to dabble in hardware, I use my C64 and C128 for real world interface and control applications. About a year ago I purchased a CCSZ Clock/ Calendar with 8K CMOS RAM (both battery backed up) from Jason-Ranheim (1805 Industrial Drive, Auburn CA, 95603).

The cartridge has proven very useful for my purposes, such as maintaining system time and protecting data integrity during power losses. The operating modes that are provided in firmware (easily called from BASIC) are too extensive to cover in this letter. The documentation is quite decent and the imaginative programmer can make the thing serve many purposes! The cost a year ago was $50 US.

Case H. Marsh
Columbia , Maryland, USA

Transbloopurz - The C128 version of "The Projector, Part II" which is listed on pages 23-25 of the January, 1988 *Transactor* does not seem to remove the hidden lines properly. On my C128 the line 1960-1970 PAINT 0 statements do not erase previously drawn lines which are a large distance below the newly drawn line.

The enclosed hidden line subroutine, if substituted for the existing lines 1950-2000 seems to work satisfactorily as a substitute. Instead of drawing only 3 lines below the most recently drawn horizontal line, the substitute program draws a sufficient number of spaced, offset lines to cover the previous lines lying below this new line. The erasure accomplished using the PAINT 0 command then erases a swath sufficiently wide to eliminate all unwanted portions of the previous line.

Lines 1851-1859 in the substitute program are intended to limit the number of offset lines to the minimum necessary to accomplish the erasure. The maximum vertical screen displacement between the most recently drawn line and its immediate predecessor is calculated for two points on each line segment connecting the precalculated and stored net points. Because these points are calculated and indexed using the integer variables x and y they do not lie along vertical lines on the screen. This is because each line is offset to the left from the previous line to achieve the 3-D illusion.

The quantity K2 represents the vertical screen distance between a precalculated point on the previous line with the interpolated point on the newly drawn line having the same horizontal screen coordinate. K3 is the same quantity using the precalculated (stored) point of the new line and the corresponding interpolated point from the previous line. If the new line lies entirely below the previous line, no lines are drawn.

I hope this may help other readers experiencing similar hidden line problems. The effort certainly helped me understand the original program more completely.

J. Milton Andres
Palos Verdes, California, USA

```
1850 rem mask hidden lines
1851 k1=0
1852 for x=0 to m-1
1853 k2=r(x,y+1)-r(x,y)+(r(x,y)-r(x+1,y))*ys/xg
1854 if k2>k1 then k1=k2
1855 k3=r(x+1,y+1)-r(x+1,y)+(r(x,y+1)-r(x+1,y+1))*ys/xg
1856 if k3>k1 then k1=k3
1857 next x
1858 k=int(k1/3)+1
1859 if k=1 then 1980
1860 for j=1 to k
1865 for i=-1 to 1
1870 locate g(0,y)+i, r(0,y)+3*j
1880 for x=1 to m
1890 draw to g(x,y)+i, r(x,y)+3*j
1900 next x, i, j
1910 locate g(0,y), r(0,y)+1
1920 for x=1 to m
1930 draw 0, +0, +0 to g(x,y), r(x,y)+1
1940 next x
1950 draw 0, +0, +0 to +8, +8
1955 for j=1 to k
1957 for i=-1 to 1
1960 paint 0, g(m,y)+i, r(m,y)+3*j
1970 paint 0, g(0,y)+i, r(0,y)+3*j
1971 next i, j
1975 for i=-1 to 1 step 2
1976 locate g(0,y)+i, r(0,y)+1
1977 for x=1 to m
1978 draw 0, +0, +0 to g(x,y)+i, r(x,y)+1
1979 next x, i
1980 locate g(0,y), r(0,y)
1990 return
2000 :
```

Cellular Automata

Mathematical Artforms For The C64 and C128

by Ian Adam, P. Eng.

There has been a great deal of mathematical exploration over the past few years of the concept of cellular automata. These are arrays of cells with specified states; each cell's state changes from generation to generation, in accordance with a fixed set of rules operating on its neighbours. The accompanying programs implement a particular variety, 4-state linear cellular automata, on the Commodore 64 and 128. You may want to use the resulting plots as an aid to academic investigation of the phenomenon; on the other hand, you can just run the program and enjoy the endlessly varying graphics it produces.

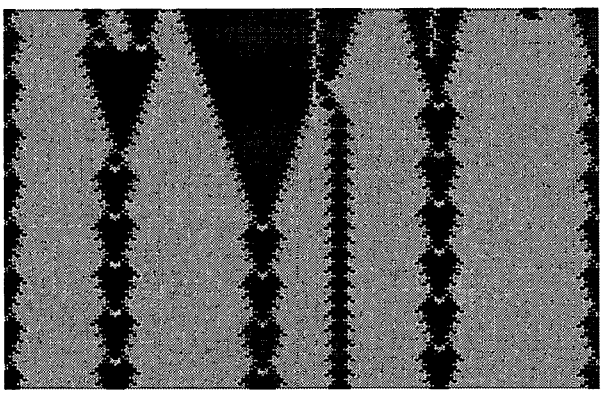


Figure 1: Code 1011303003 , seed Random

Both red and blue are capable of asserting themselves as background colours. The result is a class 2 image with intricate vertical structures. The stable structures are established quickly in spite of a random start. Cycle lengths are 7 and 23.

Background

The program, I promise you, will provide some stunning graphic images. Unlike other forms of display such as games or art programs, this one generates graphics internally in the computer, and it is amazing that applying a simple algorithm can create such a wide variety of results. These images are more than just pretty faces, however: they have brains, too. So that you can appreciate them fully, please stay tuned for some history and philosophy.

The *Encyclopedia Britannica*, 15th Edition, lists an automaton as any of various mechanical objects that are relatively self-operating once set in motion. It notes that automata are designed to arouse interest through their visual appeal, and then to inspire surprise and awe through the apparent magic of their seemingly spontaneous movement. Although written with a different application in mind, that description fits this program very well.

The earliest references to mechanical automata date all the way back to the 4th Century BC in ancient Greece and China, and include such artifacts as moving models of birds and animals. Many interesting variations have appeared over the centuries. Of particular interest is the 'magician box' of a hundred years ago: in a curious portent of the computer age, a disk engraved with a question is inserted into a slot in the box. A tiny figure of a magician then comes to life, and points with a wand to where the answer appears (on a tiny monitor, no doubt!)

More recently, the term automaton has been applied to robots and androids, and to other automatic devices that emulate human behaviour. *Collier's Encyclopedia* extends the term to include computers undertaking such human-like activities as playing chess.

By comparison, the cellular automaton has a considerably abbreviated history. The grand-daddy of modern computers, John von Neumann, began exploring self-replicating automata about 1950. His explorations led to the concept of an infinite checkerboard and a set of transition rules acting on each of its cells, resulting in a more-or-less independent machine that could transmit information, or even duplicate itself.

A popular implementation of this concept is John Conway's invention of the game of 'Life'. This game is played on an unbounded two-dimensional grid playing field; if you don't have one of those handy, you can approximate it with some sheets of graph paper. In the initial state (that is, on the first sheet of paper), some cells are 'on', or coloured in, while the rest are 'off', or blank. The next sheet of paper represents the

second generation of this lifeform, and the state of each cell is governed by a fixed rule which operates on the first generation. A new cell is 'born' if exactly three of its eight neighbours are occupied; an existing cell survives if two or three neighbours are occupied. In all other cases, the cell dies of either loneliness or overcrowding. The game was explained to the public by Martin Gardner in the *Scientific American* of October 1970. By the February 1971 issue, Gardner was already able to report on a number of interesting patterns and cycles that had been developed for this new procedure. Because the playing field is cellular, and because each cell behaves autonomously once set in motion, the term cellular automata was coined.

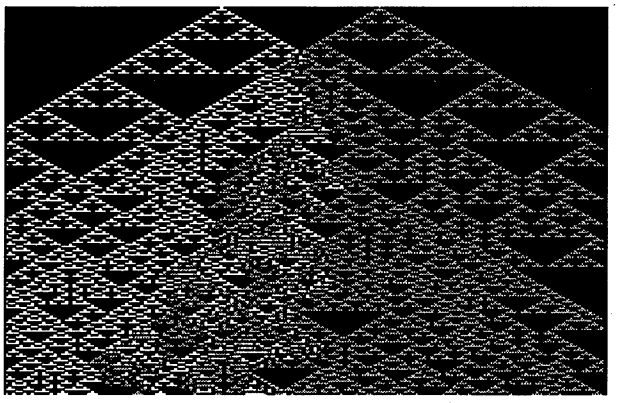


Figure 2: Code 0023010110, seed 2...3

The classic triangular branching pattern is seen at the top. A seed of 2 creates a green structure (left), while a seed of 3 give a blue structure (right). The collision area is marked by red dendritic intrusions, and blue prevails as the predominant structure.

Linear Cellular Space

Of course, many different sets of rules could be developed and applied to Conway's game - different neighbourhoods, multiple-state cells, and so on. Gardner also touched very briefly on a one-dimensional variation of cellular space, used in his application to identify a palindrome. Here the initial cells occupy one line of the graph paper; subsequent generations are usually plotted on successive lines. This results in a two-dimensional plot in which each generation's data is linear, and the second dimension (down the page) is time.

Cellular automata inhabit a simplified universe in which space is reduced to an array of cells, and time becomes a series of discrete steps. This degree of abstraction permits modelling to take place, and automata do closely resemble a number of modern computer applications. Computer chess was previously mentioned, and of course it is played on a 'bounded 2-dimensional cellular space', or checkerboard. Some other comparisons are linear, such as railroad block control and computerized traffic signals. Each of these major industrial applications uses a linear representation of its operating space, divided into cells or blocks. Vehicles are represented as 'on' cells, and their progress through the system is traced through subsequent generations by a complex set of safety rules.

These models are fairly direct and straightforward; by comparison, some of the recent interest at academic levels ranges from abstract to abstruse. Both *Scientific American* and *Nature* carried extensive discussions of cellular automata in 1984, by Brian Hayes and Stephen Wolfram. After 14 years of development, the plots were beginning to resemble some of the images you see here. Postulated applications included information processing and transmission, simulation of crystals, and a better understanding of biological processes (which are based on millions of cells each following simple rules, after all). Beyond this, suggestions that cellular automata could be applied to model languages could only be described as metaphysical at best. In December 1986, Kenneth Perry presented an algorithm in *Byte* magazine for computer display of linear cellular automata, with a more realistic main objective of creating graphics.

The linear algorithm creates a display on a high-res screen. The first row of pixels represents the initial state of the cells, each of which can have one of four values, 0 through 3, represented by different colours. As the array evolves, a new cell's state is determined by its three parents - the cells immediately above, above left, and above right. These three are added, resulting in a sum from 0 to 9.

Figure 3: Code 0023010111 , seed 3113

Only a single digit has been added to the code in photo 2, yet a much more complex image results. This includes a two-colour background and numerous short-cycle structures. Additional symmetrical patterns of considerable complexity are overlaid.

The characteristic signature of each automaton is contained in its inheritance rule, a ten-digit code that governs the evolution of cells. Each digit in the code corresponds to one value of the sum, the first digit representing a sum of 0, the second a sum of 1, and so on. For example, if the parent cells have values of 1, 3, and 2,

then the sum is 6. With a prevailing rule of '0120123123', the new cell gets a value of 3. As subsequent generations evolve, a two-dimensional plot is created on-screen (remember that the second dimension is time, not space).

The Program

I promised we'd get to this eventually, and here we are. The program includes considerable improvements over previous incarnations, probably the most important of which is speed. It is fairly easy to write a BASIC program to implement this linear cellular algorithm, particularly in BASIC 7.0. There are 200x160 multicolour points to plot, however, and about a dozen calculations for each, so the plot takes well over 15 minutes. With all the peeks and pokes of BASIC 2.0, it would take much longer. No wonder the pace of scientific progress can be so slow! Watching a universe unfold at this rate generates suspense that would make Alfred Hitchcock proud, but it is not conducive to productive research. The solution in this case is machine language, which plots the screen in less than four seconds, a considerable improvement.

The program includes a number of other features. You specify the code and the seed value for the initial generation. If you wish, either can be supplied randomly. If the plot is developing nicely, you can continue it for another screen. For when you get it just right, a simple screen dump is included.

Before you can enjoy these features, you will have to type the program in, choosing the 64 or the 128 version. Be sure to save a copy to disk before running it. Two special notes apply to the 64 version: this program modifies some pointers, and should not be saved after running. In addition, because the machine code follows BASIC, be careful not to add to its length when typing it in.

When you run the program, it starts by giving a brief description of cellular automata, and some instructions. While you are reading, the machine language is poked into memory. The program uses the digits 0, 1, 2, and 3 to correspond to black, red, green, and blue respectively. Your first input is the 10-digit code for inheritance, each digit being 0 to 3. At startup a sample code will be suggested, so just press return to accept it. In subsequent loops the previous code will be printed, which you can accept or replace with another from the table.

The next input is the seed value for the first generation. Just press R and return to get a random seed. If you want to be more selective, the procedure is different for the 64 and 128. On the 64, you enter a value for one byte, 1 to 255. This byte will be poked into position on the first row, representing 4 cells. If you enter 1, then a single cell with that value is created. A value of 255 creates 4 adjacent cells each containing 3. You also supply the column for this byte, 1 to 40.

On the 128, the seed is much more flexible. You enter a string of cell values, each 0 through 3, up to 160 digits if you wish. You also enter a pixel position 0 to 159, with 79 suggested as approximately the centre. All the cells are plotted, starting where you specify.

The Menu

As the image is drawn, 200 generations of the automaton are revealed to you. There will be a short pause to view the result, then a menu will be printed. To make a selection from the menu, just press the key shown (don't press return):

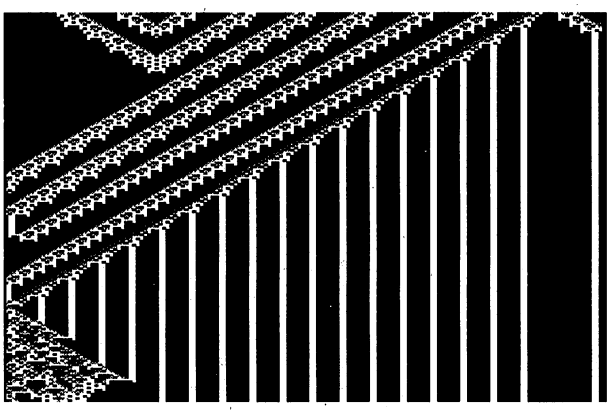


Figure 4: Code 0103220121, seed Various

Seed is selected to generate a variety of propagating structures. This code supports a variety of angled and vertical structures; one diagonal also spawns new verticals. When diagonals reach an edge or other obstacles, tthey may be absorbed, reflected, or generate other structures.

S (new seed)
You will be returned to the prompt for the seed for the first generation. The image will then be redrawn with your new seed and the same code.

R (random seed)
Random cells will be created for the first generation, and the plot will be redrawn with the same code.

M (more)
The same evolution will be drawn for another screenful (199 more generations, since the last line will become the first one displayed on the new screen).

C (new code)
You get to enter a new 10-digit code, followed by a new seed value. The new plot will then be drawn.

A (automatic)
Auto-pilot! Make this selection, and the computer will choose both random codes and random seeds. Random automata will be displayed one after another, every 4 seconds, until you press any key to get back to the menu.

P (print)
The current image will be sent to your printer, unfortunately not in colour.

Q (quit)
End the program.

Any alphanumeric key not listed here will return you to the graphic for a few more seconds, following which the menu will return.

The Screen Dump

Since it is impossible to anticipate all printer and interface combinations, this program is designed for a Cardco interface and Gemini, Panasonic, or Roland printer - popular combinations. It will work unmodified with many other systems. If it doesn't work right at first, you may be able to make some adjustments:

- Set the interface for transparent graphics mode, no line feed. Line 490 gives a secondary address of 5 to achieve this... change it to suit your interface if necessary (eg the Tymac needs a 6).

- Two commands in the program are 27,65,8 to set linefeed to 8/72 inch, and 27,75,64,1 to call for 320 graphics characters. If your printer uses different codes, change these values in line 1210.

- If after all this the printout is double-spaced, change the DATA item 10 to a zero in line 1210.

- If your printer is one of those that print each row of graphics upside down, change the value 118 in line 1230 to 54.

- Finally, if you still have a Commodore 1525 or such... my condolences. However, you will see listed some replacement lines to get a printout.

The Results

As I hope you can infer from the accompanying illustrations, the graphics images arising from these automata can be amazingly beautiful. With minor adjustments to the code, you can get images that are simple, complex, bright, sombre, or confounding. While some decay to chaos, others start from a random condition and quickly establish order. Still others seem to symbolize the eternal struggle between good and evil, never quite resolving their ultimate personality. But then, of course, we should limit ourselves to discussion of their scientific merit.

The images defy easy classification. With 4 values for each of 10 digits, there are 4g10e, or 1,048,576 possible codes. Many of these turn out to be trivial, but most produce usable results. Wolfram divides the patterns into four general groups, according to their performance after many generations:

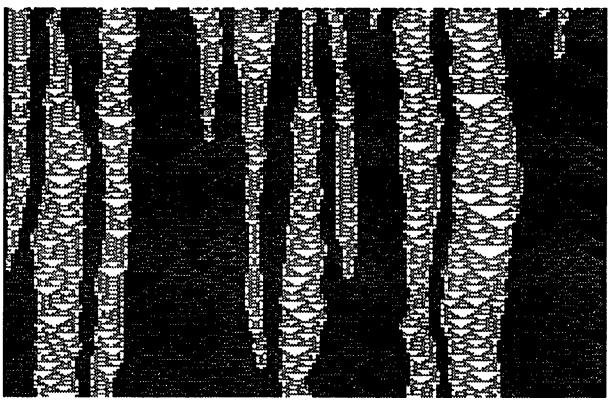


Figure 5: Code 0100132332, seed Random

A random seed produces two very distinct regimes that coexist side-by-side. This pattern is stable and is maintained through many generations. Repetitive vertical structures are also supported. There is a gradual tendency to reduce the number of distinct zones, an increase in entropy.

Class 1 automata produce plots that very quickly die out, and are basically of no interest.

Class 2 codes quickly evolve to very stable structures, mostly stripes and cyclic structures, regardless of their starting configuration (of very limited interest).

Class 3 automata produce patterns that appear to be chaotic (but note they are not random!). These structures typically grow indefinitely.

Class 4 codes have a complicated balance so that they neither grow indefinitely, nor contract and die. They generally include complex structures that are cyclical in nature, and often propagate across the field.

Perry considers only Class 4 to be worthy of further investigation: certainly these codes offer ample latitude for experimentation and mathematical study, lending themselves to considerable formalization of their structure. However, personally I find that Class 3 automata offer at least as much potential for exploration. One Class 3 code will often support several different patterns; depending upon seed values and boundary conditions, the results may range from highly ordered structures, to seemingly random textures, to cellular battlefields where rival patterns fight over territory.

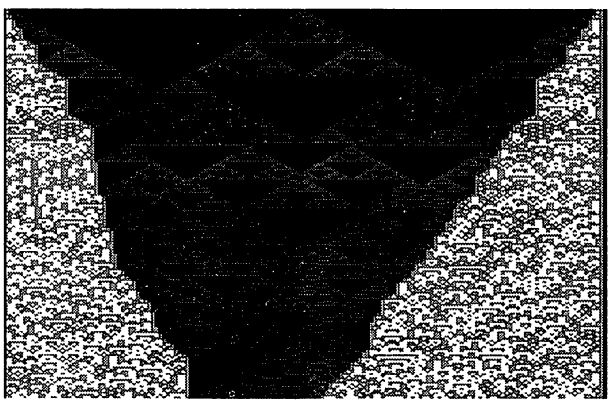


Figure 6: Code 0110133232, seed 331...1...133

This code is only slightly changed from photo 5. The blue-green regime has a very different character as a result, and now supersedes the red area in only 2-300 generations.

Because of this variety, in fact, it becomes very difficult to place these automata definitively into one category or another. For this reason, I am inclined to group them further according to the types of pattern observed, although even this is not always definitive. The table gives a number of examples of different results.

The Source Of Patterns

You will have more success creating images with a little observation of how patterns are generated. The first digit of the rule governs the background; it is commonly a zero, causing black to prevail as background. If a different digit appears, then another colour will be generated. In order for this second colour to be sustained as the background, however, it must also be able to inherit from itself. This requires that sums of 2 and 3 times the background colour also produce the same state (e.g. for green to prevail as background, the digit 2 must result from sums of 0, 4, and 6):

sum values	**0123456789**
black background	0xxxxxxxxx
red background	1x11xxxxxx
green background	2xxx2x2xxx
blue background	3xxxxx3xx3

It is apparent from this that red is at least partly compatible with the other colour backgrounds, so codes like 1011303003 will result in red and blue duelling over background rights. Blue wins in this case, but because of the fine balance between the two, a Class 2 image is created quickly from a random seed, with red structures of cycle length 7 through 23 sustained on a blue background (see figure 1).

The simplest spreading patterns are those in which a single colour propagates itself; for example, if the digit 1 appears in the code corresponding to a sum of 1, then red will spread across the field in both directions at a rate of one cell per generation. This is the fastest rate that any information can be transmitted, and is generally referred to as the 'speed of light'. Furthermore, if zeroes are present for sums of 2 and 3, then a branching algorithm is created. As the pattern spreads from a single point, it grows branches back in toward the centre. As these meet, sums of two and three are created, and their zeroes guarantee that the branches will cancel one another out. This results in the characteristic triangular pattern seen in photo 2, and occurring with so many automata. These are the minimum requirements for the branching to occur:

red triangles	0100xxxxxx
green triangles	0x2x0x0xxx
blue triangles	0xx3xx0xx0

Since the digits marked 'x' don't matter, it is apparent that there are many codes supporting this pattern, at least 4096 for each colour. Many of these will support other structures as well, leading to some interesting dual patterns. It is also clear that the green and blue patterns can co-exist with one another (in the form 0x230x0xx0). Figure 2 uses this approach, adding a touch of red where the two structures collide.

Complex Patterns

Greater complexity can be introduced through at least three measures. The first of these is the rule itself, and many examples are given in the table. Once you become familiar with the operation, you will probably want to try creating your own rules; for example, adding a single 1 to the simple code of figure 2 produces the complexity of figure 3.

Second, there is the seed, or first generation. The program will suggest starting with a single cell in the centre, but you can choose seeds more specifically in order to draw out special features from an automaton. In the alternative, a random seed will usually display many of the capabilities very quickly, and introduce a great deal of complexity in the process.

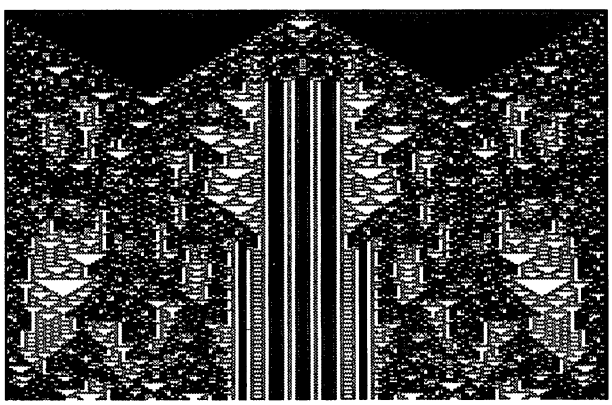


Figure 7: Code1031102332, seed 11

This automaton supports a wide variety of different patterns and structures. Black stripes, wandering red patterns and blue-green triangles all compete for territory. No clear winner can be declared after thousands of generations.

The third form of complexity is somewhat artificial, and that is the border condition. In principle at least, the automata should be viewed on an infinite field. Our screen is considerably less than infinite, being 160 pixels wide, so some mathematical impurity is inevitably introduced. Different patterns react to the border in different ways; the triangular patterns we have seen simply stop there, but even this alters the overall pattern. As shown in figure 4, some patterns will bounce or otherwise modify themselves when meeting an obstruction, and you can position the seed carefully so as to create specific effects. In most cases, however, the disturbance reflecting from the edge eventually serves to transmit chaotic conditions right across the entire field. This is a primary source of disorder in Wolfram's Class 3 automata.

In theory, this disturbance could be avoided by establishing a

pseudo-playing field in memory, much wider than the screen. A field width of, say, 1000 pixels, with only the centre portion copied to the screen, would not be a burden on the computer's RAM. This approach would permit most patterns to be followed for at least 800 generations before a reflection moving at the speed of light could return to the visible area. (This exercise is left to the reader!)

It is also left to the reader to explore the many patterns contained in the table, as well as the million or so other possibilities. I will just express my personal fascination with the many automata having two distinct regimes, of which figure 5 is but one example. Sometimes these regimes coexist peacefully side-by-side; some fight back and forth, possibly producing a victor; sometimes one pattern gradually infuses the other, rather like 'invasion of the body-snatchers'. Each is unique and fascinating.

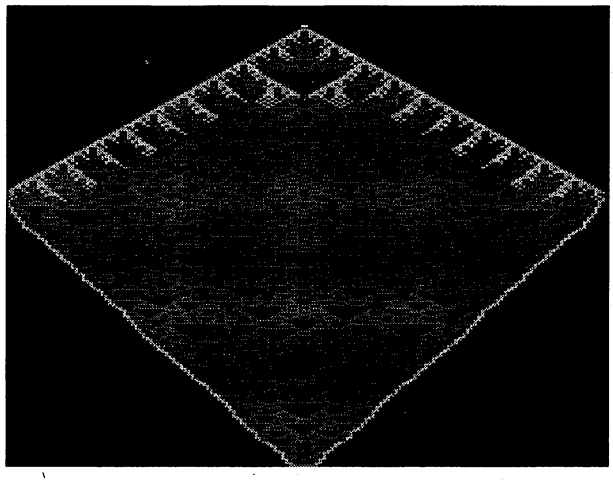


Figure 8 ("Stingray"): Code 0110310301, seed 22

A spreading blue line spawns blue bars, which in turn give rise to a red triangular pattern. This pattern is terminated by the edging, giving rise to this characteristic shape.

In Conclusion...

The patterns produced by these automata can be both fascinating and beautiful. Their beauty is more than skin deep, however, as they are valid mathematical phenomena in their own right. This program will offer you hours of experimenting and promises rewarding results, whether your interest is academic or otherwise. Send in your best results: I'd be very interested to see what you come up with. And when you get tired of all that intense experimentation... put the program on auto-pilot, relax, and enjoy the video wallpaper! As for me, I'm going to try some of these seeds in the garden, to see if I can grow an infinite crop of cellular automatoes.

A Table of Sample Rules

Code	Seed (R = Random)	Product
Class 1 (Decay quickly)		
0201023002	R	extinct in a few generations.
Class 2 (Organize to stable state)		
1011303003	R	organizes into cyclic red structures on blue (fig. 1)
0331122210	R	immediate formation of red & green stripes
0330312233	R	evolves to green & blue stripes, vertical red structures (seed 3)
0102223130	R	chaos organizes to striped wallpaper, cyclic patterns (try seed 11)
0201300003	R	evolves to long-cycle multicolour structures
0003111003	R,3,33	red/blue triangular; broad geometric patterns
2233020233	R,1,11	ice floes - blue/green blocks & patterns; rapid evolution to stripes
3032112333	R	instant self-organization as blue stripes, red/green patterns
Class 3 (Growing, chaotic)		
0023010110	2,3, 23	blue & green triangular, red dendritic in combined areas (fig. 2)
0023010111	3,R	complex trailing patterns on red & blue wallpaper (fig. 3)
0221213321	R	blue and green patterns over red blocks.
	3	same pattern, but ordered
0003232012	R,33	multicolour spreaders, green/black vertical stripes, evolves chaos
0230210313	11	blue shapes on multicolour, gradually expands for 600+ generations
0311302133	11, R	irregular red areas, blue overlay, vertical structures
3300011033	R	metallic crystal structure in red & blue.
0103002233	1,R,33	blue/black patterns, red top; blue/green stripes
1110330111	R	(a palindrome!) red/blue dendritic structure over red/black blinds
0123310203	1,3,11	blue triangles, red/green dendritic structure
3333020331	R	blue/green diagonal structures on red/black
Class 4 (Complex Structures)		
0103220121	R,11	vertical, angled structures (seeds 22,233023, 10202,2220123; fig. 4)
2221213321	3,R	blue/green vertical & angled structures on red
0201103212	3,33,R	red/green long-cycle structures on black
2001313120	R,11, 22,3	black diagonals, red diamonds, blue striped background (cover)
0300123302	1,33,101,232,3132,2331,etc	wide variety of vertical and diagonal structures, pendants
0230332101	2,22	multicolour pattern, black triangles
0210133310	R,11,2	vertical & angled structures organize into grid
0311301203	1,11,22	blue/green structures and pendants on red
0230323130	R,11,232,23303,313,etc	wintery structures, infill, pendants, etc.
0020130211	22,2,23, 322,131	red, green triangular; overlays; vertical and angled structures.

0230011133	1,11; 23 at 100	amorphous multicolour structure that may survive or not (from Wolfram)
0120133230	1; various	spreading red 'roof', spawning many blue/green structures (from Perry)

Two-regime images

0100132332	R	coexistence: red triangular; green tri's on blue; verticals (fig. 5)
0110133232	R, 232,etc	non-coexistence of same regimes (fig. 6); many variations
1031102332	R,1, 22,33	red stripes, blue/green triangles, red/black dendritic, long cycles (fig. 7)
0001231232	R	multicolour hash; black; long-cycle structures
0321200311	R,22,3	green triangular, infusion of red/blue dendritic
0233100120	R,11,22	blue triangular, infusion of red/green dendritic
1001330213	R,1, 2222	red triangles, blue dendritic infusion with green
1300122313	R,2	blue, green patches on red stripes, vertical structures
3303022133	R,33	blue & green blocks, strong diagonals
3100323120	22,1	many black & blue textures; seed 1 adds green dendritic
0120123123	1,202,R	startup screen; red/green triangular, blue/black stripes
2033210201	1,33	green jumble; red/blue stripes; some vertical structures
0132000120	2,3,1, 2003	blue, green triangular patterns, joint occupancy of space
0033220110	233,22, 222	fight for territory between blue triangular, green/red herringbone
0103310132	11,22,3	joint occupancy of red/blue wallpaper, black dendritic structure
0122022331	R,1,33	green/black triangles; blue/green regime struggles, but loses

Devolution to primordial chaos

0130312131	1 11	strong red triangle pattern, devolves to chaos retains structure, cycle length of 168 (on 64: seed 20 at positionn 19)
0113233102	11,3,33	regular stripes and patterns, decay to random
0223320100	2,3	blue or green triangular; green decays
0331210300	33,3	blue triangular, loses to red/green invasion

Suitable for framing

0110310301	22	stingray: red triangular, blue bars and edging (fig. 8).
0132320233	R,11,33	ice palace: blue blocks, green crystals
1233233320	11,R	mach waves: green vertical, red/blue stripes, blue waves (fig. 9).
0332221003	33,1,22	green/red window blinds, blue/black dendritics; eventually stable
0020133020	11	caves: green/red grid above, blue caves and triangular pattern below

More

0112002100	6201310313	1132230002	0233000001
3002110310	3103020001	0021233023	3012022322
3102033003	1200020231	3201032322	0213131022
2310131211	3011300332	3320003011	3320010231
3131120030		1302023302 (seed: 1,11,2,R)	
3132230102 (seed: 11,22)		0111212323 (seed: 1331)	
0023320103 (seed: 11,33,R)		3320012010 (seed: 33)	

• *Note: Seed values are for the 128. Here are equivalents for the 64:*

11: 5	22: 10	23: 11	33: 15	101: 17	131: 29
202: 34	222: 42	232: 46	233:47	313:55	322: 58
1331: 125	2003: 131	2222: 170	2331: 189	3132: 222	

Listing 1: "Automata 128"

```
DM  10 rem automata 128   by ian adam
EN  20 gosub 650
CJ  30 :
GM  40 do:rem main control loop
GK  50 :
CN  60 print tab(6) r$"[up]":input"code";r$
KG  70 a=5887:for i=1 to 10
DH  80 poke a+i,val(mid$(r$,i,1)) and 3
OF  90 next
IN  100 :
LM  110 print "seed value  1[left][left][left]";:input b$
MP  120 if val(left$(b$,1))=0 then 250
ND  130 a=160-len(b$)
DK  140 print "position ( 0 -"a")  79[left][left][left][left]";:input a
MM  150 graphic 3,1
CP  160 locate a,0:for i=1 to len(b$)
NB  170 draw val(mid$(b$,i,1)) and 3,+0,0
EI  180 locate +1,0:next
CD  190 :
HK  200 trap 230:sys 5900:rem plot routine
GE  210 :
AF  220 sl=200
CC  230 for i=1 to sl:if peek(208)=0 then next:print"s new seed  r
    random seed  m more":print"c new code  a automatic  p print q
    quit";:graphic 4,,23
II  240 getkey b$:print
GJ  250 on instr("srmcapq",b$) goto 280,300,340,360,380,490,610
IK  260 graphic 3:sl=700:goto 230
CI  270 :
JO  280 graphic 4,,23:goto 110:rem new seed
GJ  290 :
HF  300 r=255:graphic 3,1:rem random seed
JI  310 for i=8192 to 8504 step 8
LF  320 poke i,rnd(i)*r:next:goto 200
OL  330 :
ND  340 graphic 3:sshape s$,0,199,159,199:gshape s$,0,0:
    goto 200:rem copy last line
CN  350 :
JG  360 graphic 4,,23:loop:rem next code
GO  370 :
MK  380 graphic 3:r=255:do:r$="":rem automatic codes
GM  390 for i=5888 to 5897:a=rnd(i)*4
CJ  400 poke i,a:r$=r$+chr$(48+a):next
OA  410 :
HP  420 for i=8192 to 8504 step 8
JC  430 poke i,rnd(i)*r:next:rem seed
MC  440 :
BB  450 sys 5900
AE  460 :
CJ  470 loop until peek(208):poke 208,0:goto 230
EF  480 :
PE  490 graphic 3:open 4,4,5:
    rem secondary address = 5 for graphics, no line feed
AC  500 a$=chr$(10):cmd 4:printa$:sys 6060:printa$
LG  510 printchr$(27)chr$(64):rem reset
KG  520 printa$chr$(14)"code: "r$a$
BI  530 print#4:close 4:goto 230:screen dump
MM  600 :
GB  610 graphic 0:end
AO  620 :
PK  630 :rem start-up sequence
EP  640 :
OM  650 bank 15:color 0,1:color 1,11:color 2,14:color 3,15:color 4,1
IA  660 :
EC  670 print"[clr][white]  cellular automata for the 128
PG  680 print"[down]this program creates complex
GL  690 print"geometric artforms on the screen.
```

```
OH 700 print"the image is generated line-by-line,
OI 710 print"according to these rules:
FD 720 print"[down][yel]- a pixel has a colour value of 0,1,2,3
PH 730 print"- the values of 3 adjoining pixels
FF 740 print"  in a line are added.
DG 750 print"- the sum (0 to 9) is used to select a
KG 760 print"  new colour from the code you specify.
ME 770 print"- this new colour is plotted as the
MB 780 print"  pixel in the line below.
LG 790 print"- the code has 10 digits, corresponding
AC 800 print"  to the 10 values of the sum (0 - 9).
PD 810 print"[down][wht]automata were introduced in scientific
JM 820 print"american in 1971 & 1984, and a version
PO 830 print"appeared in byte magazine in 1986. this
GO 840 print"enhanced version for the 128 is by
LH 850 print"ian adam & transactor magazine, 1987.
DE 860 if peek(5900)-160 then gosub 1100
DO 870 print"[down][yel]  press a key!":getkey a$
EO 880 :
MA 890 print"[clr][down]  instructions:
DC 900 print"[down][wht]you enter a 10-digit rule, using only
NN 910 print"the digits 0, 1, 2, and 3.
EB 920 print"[down]next, enter a seed value, which
DH 930 print"is plotted as the top line.
AN 940 print"if you enter r, a random seed is used.
HP 950 print"if you enter seed numbers, you must
GA 960 print"also supply their position on the line.
DA 970 print"[down][yel]after plotting, press:
NO 980 print"- s  to enter a new seed
NM 990 print"- r  for a random seed
CB 1000 print"- m  more of the same plot
LE 1010 print"- c  enter new code
OL 1020 print"- a  automatic code generation
DM 1030 print"- p  send pattern to printer
BC 1040 print"- q  to quit.
JB 1050 print"[down][wht]the current code will be printed like
HC 1060 print"this. make any changes, & press return:[down]
FA 1070 r$="0120123123
EF 1080 return
GL 1090 :
EM 1100 for i=5900 to 6132:read a:poke i,a:next
CH 1110 return
EN 1120 :
OA 1130 data 160,  32, 132, 251, 132, 253, 160,   1
PK 1140 data 132, 252, 136, 132, 250, 162, 199, 134
EA 1150 data 166, 162,  39, 134, 167, 132, 169, 177
OL 1160 data 250, 133, 168, 165, 167, 240,   4, 160
KM 1170 data   8, 177, 250,  10,  38, 168,  42,  38
DN 1180 data 168,  42,  41,   3, 133, 170, 160,   4
KA 1190 data 169,   0,  38, 168,  42,  38, 168,  42
IN 1200 data  72, 101, 170, 101, 169, 170, 165, 170
CE 1210 data 133, 169, 104, 133, 170, 189,   0,  23
HN 1220 data   6, 254,   6, 254,   5, 254, 133, 254
NP 1230 data 136, 208, 221, 145, 252,  24, 198, 167
CH 1240 data  16,  40, 162,   2, 181, 250,  41,   7
KP 1250 data 201,   7, 240,  15,  56, 181, 250, 233
AM 1260 data  55, 149, 250, 181, 251, 233,   1, 149
PP 1270 data 251, 208,   6, 246, 250, 208,   2, 246
OB 1280 data 251, 202, 202, 240, 223, 198, 166, 208
DB 1290 data 144,  96, 162,   2, 181, 250, 105,   8
NH 1300 data 149, 250, 144,   3, 246, 251,  24, 202
IC 1310 data 202, 240, 241,  76,  35,  23,  27,  65
KL 1320 data   8,  13,  10,  27,  75,  64,   1,   0
KK 1330 data 160,  32, 132, 251, 160,   0, 132, 250
NM 1340 data 160,  25, 132, 252, 160,   0, 185, 162
PB 1350 data  23,  32, 210, 255, 200, 192,   9, 208
JP 1360 data 245, 160,  40, 132, 253, 160,   7, 177
FG 1370 data 250, 162,   7,  42, 118, 166, 202,  16
HM 1380 data 250, 136,  16, 243, 169,   7, 170,  56
KF 1390 data 101, 250, 133, 250, 144,   2, 230, 251
EM 1400 data 181, 166,  32, 210, 255, 202,  16, 248
BO 1410 data 198, 253, 208, 217, 198, 252, 208, 196
FB 1420 data  96
```

Listing 2: "Automata 64"

```
KM 10 gosub 650
CC 20 automata 64   by ian adam
CJ 30 :
LH 40 : main loop
GK 50 :
CN 60 print tab(6)r$"[up]":input"code";r$
KG 70 a=5887:for i=1 to 10
DH 80 poke a+i,val(mid$(r$,i,1)) and 3
OF 90 next
IN 100 :
LM 110 print "seed value  1[left][left][left]";:input b$
CG 120 if val(b$)=0 then 250
LN 130 print "position (1-40)  20[left][left][left][left]";:input a
PH 140 gosub 540:for i=8192 to 8504 step 8:poke i,0:next
GL 150 poke 8184+8*a,val(b$) and 255
EB 160 :
JP 170 sys 5900
IC 180 :
PL 190 for i=0 to 999:if peek(k) then 240
MN 200 next:gosub 590:print"m more of this  p print this
DF 210 print"s new seed  r random seed
GN 220 print"c new code  a automatic codes
NG 230 print"v view plot  q quit";
PF 240 wait k,7:get b$
CL 250 for i=1 to 7:if mid$("srmcapq",i,1) <> b$ then next
LI 260 on i goto 290,310,330,370,390,490,610
DD 270 gosub 540:goto 190
MI 280 :
FE 290 gosub 590:goto 110:new seed
AK 300 :
CK 310 r=255:def fns(x)=rnd(x)*r:goto 340:random seed
EL 320 :
FM 330 r=7687:def fns(x)=peek(r+x):rem copy last line
HK 340 for i=8192 to 8504 step 8
HE 350 poke i,fns(i):next:gosub 540:goto 170
MN 360 :
CG 370 gosub 590:goto 60:new code
AP 380 :
MO 390 gosub 540:r=255:for j=0 to 1:r$="":rem automatic
CO 400 for l=5888 to 5897:a=rnd(l)*4
FK 410 poke l,a:r$=r$+chr$(48+a):next
IB 420 :
KA 430 for l=8192 to 8504 step 8
DB 440 poke l,rnd(l)*r:next
GD 450 :
LB 460 sys 5900
BC 470 j=peek(k):next:poke k,0:goto 190
EF 480 :
GK 490 gosub 540:a$=chr$(10):open 4,4,5
      :rem 2nd addrs grafix, no lf
EF 500 cmd 4:sys 6060
OD 510 printa$a$chr$(14)"code: "r$a$a$
JM 520 print#4:close 4:goto 190
GI 530 :
LN 540 if peek(v)=59 then return
MC 550 poke v,59:poke v+5,216:poke v+7,24:rem hires
PP 560 print"[home][lt. blue]";:for i=1 to 111:print
      "[rvs]---------";:next
DK 570 poke 2023,173:poke 56295,14:return:colors
IL 580 :
IN 590 poke v,27:poke v+5,200:poke v+7,21:print:return:text
MM 600 :
```

```
LL  610 sys 65409:end
AO  620 :
FL  630 : start-up
EP  640 :
FB  650 poke53280,0:poke53281,0:poke46,64:clr:k=198:v=53265
IA  660 :
KM  670 print"[clr][white]  cellular automata for the 64
PG  680 print"[down]this program creates complex
GL  690 print"geometric artforms on the screen.
KP  700 print"the image is generated line-by-line
OI  710 print"according to these rules:
HF  720 print"[down][yellow]- a pixel has a colour value 0,1,2,3
LC  730 print"- add the values of 3 adjoining pixels
KN  740 print"  in a line.
LF  750 print"- the sum (0-9) is used to select a
KG  760 print"  new colour from the code you specify.
ME  770 print"- this new colour is plotted as the
OI  780 print"  pixel directly below.
LG  790 print"- the code has 10 digits, corresponding
AC  800 print"  to the 10 values of the sum (0-9).
PD  810 print"[down][white]automata were introduced in scientific
JM  820 print"american in 1971 & 1984, and a version
PO  830 print"appeared in byte magazine in 1986. this
MF  840 print"enhanced version for the 64 is by
LH  850 print"ian adam & transactor magazine, 1987.
DE  860 if peek(5900)-160 then gosub 1100
PE  870 print"[yellow]press return!":input a$
EO  880 :
MA  890 print"[clr][down]  instructions:
AP  900 print"[down][white]you enter a 10-digit rule, using
NN  910 print"the digits 0, 1, 2, and 3.
BI  920 print"[down]next, enter a seed value which
HH  930 print"is plotted on the top line.
AN  940 print"if you enter r, a random seed is used.
EI  950 print"if you enter a seed #, you must
NH  960 print"also supply its position on the line.
DA  970 print"[down][yellow]after plotting, press:
IO  980 print"- s  enter a new seed
NM  990 print"- r  for a random seed
CB 1000 print"- m  more of the same plot
LE 1010 print"- c  enter new code
OL 1020 print"- a  automatic code generation
DM 1030 print"- p  send pattern to printer
GM 1040 print"- q  quit
BE 1050 print"[down][white]the current code will be shown like
JO 1060 print"this. make any changes & press return:[down]
FA 1070 r$="0120123123
MK 1080 goto 60
GL 1090 :
EM 1100 for i=5900 to 6132:read a:poke i,a:next
CH 1110 return
EN 1120 :
OA 1130 data 160,  32, 132, 251, 132, 253, 160,   1
PK 1140 data 132, 252, 136, 132, 250, 162, 199, 134
EA 1150 data 166, 162,  39, 134, 167, 132, 169, 177
OL 1160 data 250, 133, 168, 165, 167, 240,   4, 160
KM 1170 data   8, 177, 250,  10,  38, 168,  42,  38
DN 1180 data 168,  42,  41,   3, 133, 170, 160,   4
KA 1190 data 169,   0,  38, 168,  42,  38, 168,  42
IN 1200 data  72, 101, 170, 101, 169, 170, 165, 170
CE 1210 data 133, 169, 104, 133, 170, 189,   0,  23
HN 1220 data   6, 254,   6, 254,   5, 254, 133, 254
NP 1230 data 136, 208, 221, 145, 252,  24, 198, 167
CH 1240 data  16,  40, 162,   2, 181, 250,  41,   7
KP 1250 data 201,   7, 240,  15,  56, 181, 250, 233
AM 1260 data  55, 149, 250, 181, 251, 233,   1, 149
PP 1270 data 251, 208,   6, 246, 250, 208,   2, 246
OB 1280 data 251, 202, 202, 240, 223, 198, 166, 208
DB 1290 data 144,  96, 162,   2, 181, 250, 105,   8
NH 1300 data 149, 250, 144,   3, 246, 251,  24, 202
IC 1310 data 202, 240, 241,  76,  35,  23,  27,  65
KL 1320 data   8,  13,  10,  27,  75,  64,   1,   0
KK 1330 data 160,  32, 132, 251, 160,   0, 132, 250
NM 1340 data 160,  25, 132, 252, 160,   0, 185, 162
PB 1350 data  23,  32, 210, 255, 200, 192,   9, 208
JP 1360 data 245, 160,  40, 132, 253, 160,   7, 177
FG 1370 data 250, 162,   7,  42, 118, 166, 202,  16
HM 1380 data 250, 136,  16, 243, 169,   7, 170,  56
KF 1390 data 101, 250, 133, 250, 144,   2, 230, 251
EM 1400 data 181, 166,  32, 210, 255, 202,  16, 248
BO 1410 data 198, 253, 208, 217, 198, 252, 208, 196
FB 1420 data  96
```

Listing 3: "Lines for 1525"

```
GA  1 rem lines for 1525
GL  490 gosub 540:a$=chr$(10):open 4,4:rem cbm 1525
GM 1100 for i=5900to6134:read a:poke i,a:next
CF 1310 data 202, 240, 241,  76,  35,  23,  13,   8
GP 1320 data   0,   0,   0,   0,   0,   0,   0,   0
IB 1350 data  23,  32, 210, 255, 200, 192,   2, 208
MF 1370 data 250, 162,   7,  42,  54, 166, 202,  16
KP 1400 data 181, 166,   9, 128,  32, 210, 255, 202
CB 1410 data  16, 246, 198, 253, 208, 215, 198, 252
OP 1420 data 208, 194,  96
```

Listing 4: "automata.src"

```
KB  100 ; ********************
MJ  110 ; **                  **
PH  120 ; **    cellular      **
PL  130 ; **    automata      **
KL  140 ; **                  **
ME  150 ; ********************
GB  160 ;
AC  170 ;
KC  180 ;
PN  190 ; geometric  computer
CN  200 ;     artforms
IE  210 ;
HP  220 ;     for  the
PO  230 ; commodore  64 & 128
GG  240 ;
AA  250 ;   by  ian  adam
GL  260 ;   vancouver  bc
EI  270 ;
LD  280 ;   march   1987
IJ  290 ;
CK  300 ;
FB  310 ; the screen image is plotted 1
CM  320 ; line at a time.  each pixel
PJ  330 ; depends on the sum of the 3
NJ  340 ; pixels above, using a preset
FH  350 ; code supplied by the user.
ON  360 ;
JC  370 zp      =$a6     ;8 bytes temporary
GM  380 rows    =$a6
FB  390 column  =$a7
OJ  400 bits    =$a8
EI  410 aval    =$a9
IK  420 bval    =$aa
KF  430 adread  =$fa     ;read address
JF  440 adwrit  =$fc     ;write address
MO  450 output  =$fe
MA  460 screen  =$2000
IJ  470 bsout   =$ffd2
GF  480 ;
MN  490 *= $1700         ;same for both
```

```
KG 500  ;
CG 510  codes       *= *+10     ;these are the rules
IA 520                          ; for plotting pixels (10 bytes)
II 530  ;
GO 540  ;************************
EF 550  ;**                     **
CB 560  ;** start plotting here **
IG 570  ;**                     **
OA 580  ;************************
EM 590  ;
BI 600  *= $170c                ;friendly address (5900)
IN 610  ;
AE 620  ; set up pointers
MO 630  ;
NN 640          ldy  #>screen   ;set addresses
HM 650          sty  adread+1
DG 660          sty  adwrit+1
FC 670          ldy  #$01
KI 680          sty  adwrit     ;write to $2001
NG 690          dey
BM 700          sty  adread     ;read $2000
MD 710  ;
EK 720          ldx  #$c7
DK 730          stx  rows       ;199 rows to do
KF 740  ;
IA 750  ; setup for each row
OG 760  ;
GB 770  startr   ldx  #$27      ;40 bytes per line
AB 780          stx  column
MI 790  ;
JC 800  ; aval is pixel above & left
OE 810  ; bval represents pixel above
LP 820  ; cval is pixel above & right
EL 830  ;
OF 840          sty  aval       ;aval = 0 to start row
IM 850  ;
JN 860  ; prepare one byte at a time
MN 870  ;
ME 880  startc   lda  (adread),y ;get byte above
MD 890          sta  bits
KP 900  ;
IP 910          lda  column
LH 920          beq  get1st
CP 930  ;note: we need the first pixel
ND 940  ;from the next byte to the right,
BA 950  ;to be cval for the 4th pixel of
BE 960  ;this byte. on the last screen
IL 970  ;block of a row, counter 'column'
GI 980  ;will be zero.  in this case,
EF 990  ;a 0 will be put into variable
PE 1000 ;cval for the last pixel in the
FL 1010 ;row. if not the last block, then
PO 1020 ;get a pixel from the next block:
MH 1030 ;
LG 1040         ldy  #8
AK 1050         lda  (adread),y
KJ 1060 ;
PI 1070 get1st   asl  a
FA 1080         rol  bits       ;extra pixel into bits
EE 1090         rol  a
JC 1100         rol  bits
DN 1110         rol  a          ;and 1st pixel rolls
DA 1120         and  #3         ;into a, then
GP 1130         sta  bval       ;...into bval
KO 1140 ;
CA 1150 ; pixel loop for one byte
OP 1160 ;
GF 1170         ldy  #4         ;4 pixels
AH 1180 pxloop   lda  #0
BB 1190         rol  bits       ;get one pixel
CL 1200         rol  a
HJ 1210         rol  bits
GM 1220         rol  a
IC 1230         pha             ;this pixel is cval
OE 1240 ;
CO 1250         adc  bval       ;form sum of 3 pixels
PF 1260         adc  aval       ;(carry is clear)
BM 1270         tax
GH 1280 ;
BC 1290         lda  bval       ;shift records over
FI 1300         sta  aval
JL 1310         pla             ;get cval back
KJ 1320         sta  bval
IK 1330 ;
NF 1340         lda  codes,x    ;get new colour value
KC 1350         asl  output     ;make room in byte, &
NC 1360         asl  output
HA 1370         ora  output     ;put pixel in stream
MD 1380         sta  output
EO 1390 ;
IJ 1400         dey             ;move to next pixel
DB 1410         bne  pxloop
CA 1420 ;
DL 1430 ; finished pixel loop for
HL 1440 ; byte, so output the result:
AC 1450 ;
OI 1460         sta  (adwrit),y
ED 1470 ;
PO 1480 ; update addresses:
IE 1490 ;
CF 1500         clc
BD 1510         dec  column     ;where on screen?
MI 1520         bpl  oldrow
AH 1530 ;
MA 1540 ; here because end of row, so
EI 1550 ;
PH 1560 ; update pointers to start next row
IJ 1570 ;
PF 1580         ldx  #2         ;do adwrit first
LM 1590 newrw1  lda  adread,x
DO 1600         and  #7
JL 1610         cmp  #7         ;check if bottom of block
ID 1620         beq  newlin
EN 1630 ;
GF 1640         sec             ;next pixel row, subtract 311
AE 1650         lda  adread,x
EN 1660         sbc  #$37
CJ 1670         sta  adread,x
PI 1680         lda  adread+1,x
HJ 1690         sbc  #1
BO 1700         sta  adread+1,x
DE 1710         bne  newrw2
OC 1720 ;
LF 1730 newlin   inc  adread,x   ;if bottom of
PB 1740         bne  newrw2     ;block, just add 1
FP 1750         inc  adread+1,x
GF 1760 ;
FM 1770 newrw2  dex             ;now do adread
LK 1780         dex
EK 1790         beq  newrw1
OH 1800 ;
NH 1810         dec  rows       ;ready for next row
MJ 1820         bne  startr
CB 1830         rts
GK 1840 ;
OB 1850 ; here because in middle of row
KL 1860 ;
KK 1870 ; so move to next byte
```

```
OM 1880 ;
PB 1890 oldrow  ldx  #2          ;start with adwrit
CO 1900 ;
MO 1910 oldrw2  lda  adread,x    ;mid column so
LN 1920         adc  #8          ;move over one byte
GJ 1930         sta  adread,x
FK 1940         bcc  oldrw3
NL 1950         inc  adread+1,x
OB 1960         clc
IC 1970 ;
MO 1980 oldrw3  dex
NH 1990         dex
JG 2000         beq  oldrw2      ;now do adread
MI 2010         jmp  startc      ;start next column
KF 2020 ;
EG 2030 ;
KL 2040 ; ************************
AD 2050 ; **                    **
DI 2060 ; **   screen dump here **
EE 2070 ; **                    **
CO 2080 ; ************************
AK 2090 ;
FO 2100 rowout  =    adwrit
HG 2110 colout  =    adwrit+1
OL 2120 ;
JO 2130 messag  .byt 27,65,8,13,10,27,75,64,1
CN 2140 ;
KM 2150 ; 27,65,8    sets graphics linefeed
HH 2160 ; 13,10      is carriage return & lf
CK 2170 ; 27,75,64,1 for 320 graphics bytes
KD 2180 ; change these for other printers
EA 2190 ;
OA 2200 ;
FC 2210 ; printer is already accessed as
MF 2220 ; CMD file by BASIC program
MC 2230 ;
GD 2240 ;
BP 2250 *= $17ac                 ;6060 is a friendly start
KE 2260 ;
PF 2270 ; set up pointer
OF 2280 ;
EB 2290         ldy  #>screen    ;set screen address
JD 2300         sty  adread+1
JI 2310         ldy  #$00
EH 2320         sty  adread
EM 2330         ldy  #$19
GJ 2340         sty  rowout      ;25 rows to do
EK 2350 ;
HD 2360 ; set up for row of 320 bytes
IL 2370 ;
CE 2380 oprow   ldy  #0
AH 2390 linmsg  lda  messag,y
CL 2400         jsr  bsout
KE 2410         iny
DO 2420         cpy  #9
NN 2430         bne  linmsg
OP 2440 ;
BC 2450         ldy  #$28        ;output 40 columns
AK 2460         sty  colout
MB 2470 ;
LA 2480 block   ldy  #7          ;one block of 8 bytes
OK 2490 bytelp  lda  (adread),y
KD 2500 ;
DN 2510 ; reorient bytes 90 degrees
OE 2520 ;
KD 2530 ; screen bytes are horizontal
PG 2540 ; printer bytes are vertical
MG 2550 ;
FF 2560         ldx  #7
EO 2570 rotate  rol  a           ;one bit into each
DH 2580         ror  zp,x        ; of 8 bytes
EA 2590 ;change ror to rol if your
OF 2600 ;printer does graphics inverted
IK 2610 ;
DP 2620         dex
IL 2630         bpl  rotate
GM 2640 ;
FB 2650         dey
IP 2660         bpl  bytelp
EO 2670 ;
JI 2680 ; move pointer 8 bytes
DP 2690 ; for next screen block
CA 2700 ;
PI 2710         lda  #7
LG 2720         tax
LC 2730         sec
MG 2740         adc  adread
CM 2750         sta  adread
PB 2760         bcc  oploop
BJ 2770         inc  adread+1
CF 2780 ;
CO 2790 ; output 8 bytes
GG 2800 ;
FL 2810 oploop  lda  zp,x
GF 2820         jsr  bsout
FM 2830         dex
KL 2840         bpl  oploop
IJ 2850 ;
PO 2860 ; update counters
MK 2870 ;
MF 2880         dec  colout      ;next column
KO 2890         bne  block
KM 2900 ;
NJ 2910         dec  rowout      ;next row
CG 2920         bne  oprow
IO 2930 ;
IG 2940         rts
MP 2950 ;
AD 2960 ; BASIC takes care of unlistening
FA 2970 ; and closing printer file.
KB 2980 ;
KI 2990 .end
```

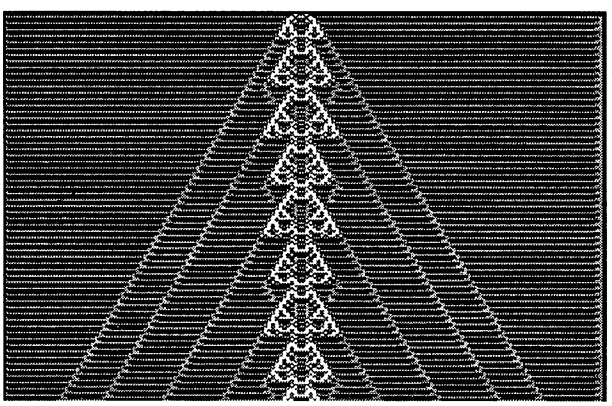


Figure 9 ("Mach Waves"): Code 1233233320, seed 11

The background consists of red and blue lines. The repeating green central pattern creates a series of blue interference waves that create a strong image. The pattern continues indefinitely.

CP/M Plus + CoNIX = CP/M Plus+

A CP/M enhancement with a Unix flavour

by Adam Herst

The 'Plus' in CP/M Plus holds the promise that this version of CP/M is more than just a retread of the time-worn operating system. It suggests the presence of new ideas and enhanced capabilities. But the reality is a mere shadow of the ideal.

CoNIX, an operating system enhancement for computers running CP/M-80, makes good on the promise of CP/M Plus. The combination of the C-128, CP/M Plus and CoNIX proves a powerful combination. The CoNIX package implements a programming environment around CP/M that in many ways surpasses that of CP/M's supplanter, MS-DOS. Its name suggest comparisons with the popular minicomputer operating system, Unix. After working with CoNIX these last few months, I can say that the comparison is apt.

CoNIX is not a CP/M replacement. Instead, it adds functions and capabilities to the operating system, maintaining compatibility for existing CP/M programs. No aspect of CP/M operation is left unenhanced. New system calls, more command line utilities, a system level Command Language and a library of utilities written in the Command Language are just some of CoNIX's features.

CoNIX was developed and is distributed by Computer Helper Industries Inc. CHI distributes the CoNIX environment in three packages: the CoNIX Operating System, the CoNIX Programming System and the CoNIX Library of XCC Utilities (XCC is the CoNIX Command Language 'interpreter'). However, the three packages are intimately intertwined in their operation and the divisions between them appear to be mostly in name. Nonetheless, for the purposes of this article, the distinction between the packages will be maintained.

CoNIX: The Operating System

The heart of CoNIX is the CoNIX Operating System. Neither of the other packages can be run without it. CoNIX is called an operating system because "...it is in total control of all system hardware and software, and all programs must pass through it when they are running."

CoNIX replaces the CP/M Console Command Processor (CCP) with the CONIX.COM program. It is through this new command processor that CoNIX is able to provide scores of internal utilities, a customizable user environment, additional command line functionality, an assortment of variables, and enhanced file management.

The CCP provided with CP/M Plus comes with a small number of internal or resident commands. The CoNIX command processor has over 20 internal commands, including all the functions provided in the CCP of CP/M Plus. Some of these are:

BDEC and DECB	convert binary values to decimal values and vice versa
CHR	convert ASCII values to hex
ECHO	print the arguments to screen
EXAM	examine memory
FILL	fill memory
FIND	find file in the search path
FLUSH	empty the print spooler
INDEX	find a string in memory
MOVE	copy memory
OPT	set environment option
UDIR	list files through user areas
WRITE	write memory to a file
ZAP	modify memory

You may have noticed a few utilities that perform functions that have no place in a normal CP/M system. Flushing the print spooler? Searching the file path? Setting the environment? All of these are possible with CoNIX.

The most powerful of the utility commands is OPT. It allows the customization of most of the capabilities of CoNIX. (The number of definable fetures is too great to cover exhaustively

- only those I use most will be mentioned.) A print spooler can be enabled on any disk up to the maximum size of the disk. Printer output is sent to the printer during keyboard polling. The spooler can be flushed and overridden. Other customizable options include the specification of sizes and locations of internal stacks, buffers and pointers, the location of temporary files and data files used by the CoNIX system, the definition and enabling of path searching, and the default setting for memory management. These last two items deserve further elaboration.

CoNIX improves on the file manipulation capabilities of CP/M Plus in several ways. User areas are more accessible - a program in any user area can now be executed from any other user area. File location is designated with the syntax:

D:U/

where D is the drive letter and U is the user area number. The commands and utilities provided with CoNIX accept this syntax for their arguments, allowing access to data files in other user areas as well as command files.

When a command is issued under CP/M Plus, both the current user area and user area 0 of the default drive, the only locations from which files can be executed, are searched. CoNIX extends the search path to include any drive or user area. There is no limit to the extent of the search. To allow non-CoNIX programs to find their overlay, help and other run-time files, a list of file extensions in addition to .COM files can be added to the search. To allow programs to look for their data files in other user areas in the search path automatic file searching can be invoked. (This requires the data file name to be prepended with a colon, thereby reducing the effective file name length to seven characters.) Finally, the CoNIX environment is equipped with an archive manager, ARM.COM. This program collects many files into a manageable single file, reducing disk storage overhead, a major problem with CP/M. Commands can be executed directly from these archives, which can be added to the search path.

Finally, one environment option sets the default memory management level. This option defines the 'level' of the CoNIX program that remains resident in memory. As you may have guessed from the functionality provided by CoNIX, it is not a small program - 28K to be exact. If all of the CoNIX program were to remain in memory at all times, it would leave little room for the execution of other programs. To avoid this restrictive condition, the CoNIX program has been divided into a number of functional levels. The full 28K is used when all of the levels are resident. A minimal 1/2K is used when the lowest level is resident. CoNIX functionality decreases with the number of levels resident in memory. The default memory management level can be set with the OPT command and individual commands can set the memory management level for the duration of their execution.

As the primary interface to CoNIX, and by extension to CP/M, the CoNIX command processor provides great freedom and variety in the forms of allowable input. A particularly useful example is the use of the backslash (\) as a mask or 'non-interpret' character. This allows all ASCII characters to be entered at the command line, even those with assigned special functions. Other examples include a variety of character case mappings and data type conversions.

CoNIX provides a variety of variables, all accessible at the command line. These include: disk-based variables, hexadecimal variables and memory variables.

The 52 disk-based variables are referenced as $a to $z and $A to $Z. They are set with the internal SET command and their values stored in a disk file. When one of these variables is used, the disk file is read and the value substituted. These variables each can hold strings of up to 255 characters in length, including references to other variables.

The 16 hexadecimal variables are referenced as $$0 to $$F. They are used primarily to pass values to and from resident commands. They are pivotal in the execution of CoNIX Command Language programs.

The most interesting (and potentially most useful) variables provided by CoNIX, are the memory variables. Memory variables are referenced by a $@ sign followed by a 16-bit hexadecimal address. The contents of memory starting at that address, and usually terminated by an FF (this, as with so many other features of CoNIX, is user definable), is then substituted. If no address is given, the contents of CoNIX's internal, 128 byte memory buffer is used. As may have been guessed, the number and size of memory variables are system and application dependent.

Through its command processor CoNIX also provides a very rich implementation of I/O redirection. In a recent article I talked about the PUT and GET commands of CP/M Plus and complained that they were non-standard and 'untrue' implementations of redirection. CoNIX provides true I/O redirection for both devices and files. Input and output can be redirected, respectively from and to the 'raw' console keyboard, the 'null' device, the console keyboard, a user defined device, a user defined memory address and a memory 'file', as well as the expected disk file.

In addition, printer output can be redirected to other devices, memory files or disk files. A variety of command-line redirection options are available to control and process the data stream. Finally, CoNIX implements command PIPES (the direct use of the output of one command as the input to a second command), the logical extension of redirection. It is one of the few microcomputer operating systems to do so.

CoNIX: The Programming Language

Earlier I said that the divisions between the three packages in the CoNIX environment appear to be abitrary. This is most

pronounced with the division between the CoNIX Operating System and the CoNIX Command Language. The CoNIX Command Language is little more than a programming manual and the 'interpreter' to turn CoNIX Command Language programs into .COM files, executable only under CoNIX. All of the commands used in the programs are available under the CoNIX Operating System. However, without the manual to tell you what they are, and the XCC program to turn them into runnable form, these commands are useless. The value of documentation to today's complex programs should not be underestimated.

The CoNIX Programming Manual details the use of the XCC interpreter, the flow of control commands, the many programming variables (yes, Virginia, there are more variables), the operating system command line options (mentioned only in passing in the Operating System Instruction Manual), the general programming commands, the programming utilities and the added CoNIX system calls.

The CoNIX Command Language provides the facilities of any structured programming language. Unlike most programming languages, which are designed to operate in isolation, the CoNIX Command Language is designed to interact with the operating system and the programs and commands which run under it. Used simply, the CoNIX Command Language can automate repetitive tasks, similarly to the CP/M Plus SUBMIT command that it replaces. Used to its fullest, the CoNIX Command Language can join disparate and distinct commands and programs into new and unique software tools. The CoNIX Command Language contains the flow of control constructs expected in modern programming languages. Simple conditional evaluations (AND and OR) are possible. More complex conditional evaluations are possible with IF-THEN-ELSE and SWITCH constructs. Branching is possible using any of GOTO, GOSUB or WHILE constructs. These commands are accessible only through CoNIX Command Language programs. All appropriate constructs can be nested (it's hard to nest a GOTO) to a default value of 255 levels. As with most other CoNIX parameters, these values can be individually tailored to suit your needs within the restrictions of your system's resources. Each of the constructs has an associated command to break out of any specified number of nested levels.

Construct tests are based on the exit status of commands and programs. Only CoNIX commands and user-written programs designed to run under CoNIX will set the exit status. Only these programs can be used directly in construct tests. However, other methods, outlined in the documentation, exist for the indirect use of standard CP/M commands and programs.

An assortment of programming variables is available for use with CoNIX Command Language programs. Command Line Argument Variables, referenced as $0 to $255, allow the passing of parameters to Command Language programs. Memory Address Variables, referenced as $&<address>, where <address> is a hexadecimal memory address, allow the manipulation of two-byte data anywhere in memory. Finally, Environment Variables allow for the testing and monitoring of many system functions. These include: the BDOS error status, the default disk drive, the current user area, the end of file status, the current nest level, the column position of the cursor on the screen, the column position of of the last character output on non-screen devices and, of course, the exit status.

Included in the Command Language manual is a chapter on the more than twenty programming commands. The introduction to the chapter states that these commands are accessible on the operating system command line as well as in Command Language programs. If you had bought only the operating system, you would never know they were there.

The programming commands generally fall into two categories by function: system interface and string manipulation. Examples of the system interface commands are:

BDOS	execute a system call or accessible user routine with the loading of registers
FNAME	process a filename into its components
PUSH, POP	push and pop strings onto and off a user-defined stack in memory

The string processing commands are greater in number and include:

GETC, GETL	read a character/line from the standard input
ISC, ISN, ISX	check if a string is a character, numeric or hexadecimal string, respectively
LEN	print a string length
SCMP	compare strings
SUBSTR	return a substring
STRIP	strip leading characters from a string
SUM, SUB	add and subtract two numbers
TEST	test two numbers for equality

Additional programming utilities, omitted from CoNIX proper to minimize program size, are distributed with the CoNIX Command Language. Only two will be mentioned here. Of most general use is the utility program EXPR, an expression analyzer. The CoNIX command lnaguage, oriented towards string processing, performs only the most rudimentary mathematical operations. EXPR can be used to supplement these resources when the need arises.

The second utility is MKREL - make a relocatable program. Relocatable programs are one part of another great idea from the developers of CoNIX. Using MKREL, and following a simple prescribed methodology, user programs can be written that load into and execute from any point in memory. Since programs normally load into memory at 100h, loading a program causes the previous one to be displaced from memory.

To repeatedly execute a program requires that it be repeatedly loaded from disk. Storing and executing multiple relocatable programs promises a significant reduction in disk I/O and its associated overhead.

Supporting the many features of CoNIX are 23 new system calls. These system calls are documented in the CoNIX Command Language Manual and are accessible to user written programs. From the description of the CoNIX operating system and Command Language, you can imagine the breadth of the new system calls. I won't list them. Obviously, programs accessing these calls will not run under standard CP/M.

The CoNIX commands, constructs and utilities described above, along with any user commands or programs, can be brought together in Command Language programs. Command Language programs are ASCII text files, prepared with any text editor, typically with the file extension .xcc. The XCC Command Language Interpreter must be used to turn the source files into eXecutable Command Code, with an extension of .com. Error checking is performed during 'compilation' and a number of XCC debugging options are accessible through the command line.

While XCC programs have a .com extension, they are not like regular .com files. First, they will not run on a standard CP/M system. Second, unlike standard .com files, XCC programs do not load into memory to execute. Instead, "...execution takes place on disk, with CoNIX reading 128-byte records into an internal area of memory from which the program is processed". This allows XCC programs to be as large as available disk space, removing program size limits imposed by system memory.

CoNIX: The XCC Library

The CoNIX programming environment is as sophisticated and versatile as that found on many mini and mainframe computers. As such it presents a foreign and potentially frustrating environment for new users. The CoNIX Library of XCC Utilities, distributed as ready to run programs, include the XCC source code as tutorial examples of XCC programming. The printouts of these programs total over 100 pages and reveal many of the tricks of XCC programming.

The functions of some of the XCC utilities are worth mentioning in themselves. Using XCC programs, CoNIX implements a system of hierarchical directories and provides a complement of utilties to manipulate the file system. These include utilities to make and remove directories, list directory paths and file contents, move, copy and link files across directories and a shell to process path names for other programs. All this is performed through the manipulation of text files by the XCC programs. There is an I/O overhead from the extra disk access but the system performs surprisingly well.

Other utilities are an interactive file un-erase utility, a utility to do simple formatting of a file and send it to the printer, a file display utility, and more. Space constraints prevent a full description of these but suffice it to say that many of them singly are worth the price of the XCC Library package in total.

CoNIX: The Documentation

Each of the three CoNIX packages, Operating System, Programming Language and XCC Library, comes with a plastic-spiral bound manual - in total almost a rival in size to the Digital Research Inc. CP/M Plus manual. Chapters are well organized and, more importantly, well written. Concepts are presented from first principles. Little, if no, prior knowledge of operating systems or programming languages is assumed. Examples abound. Each manual has a comprehensive index. A truly professional attitude is evident throughout - a quality all too often lacking from computer software documentation.

If all of the commands and options sound like too big a handful to keep hold of, and the documentation too cumbersome to use with your hands full, on-line help and a simple but effective menu program to configure the CoNIX environment are included in the package. These files require a lot of disk space and are best stored in the C-128's RAM disk if you expect to receive help in real time. This overhead makes the help systems impractical most of the time, but in the first few weeks of using CoNIX they will be the first files you load.

CoNIX: The Support

CoNIX and Computer Helper Industries are an oasis in the CP/M software desert. Finding support for most CP/M software packages is an insurmountable problem. Manufacturers have either gone out of business or (and this is the case with the manufacturer of CP/M Plus, Digital Research Inc.) have discontinued support for the product. CoNIX, whose current version is numbered 22.x, evolves with the computers it can run on. A call to Computer Helper Industries (at my expense - they do not provide toll free service) yielded a speedy fix to my bug report (they called me back) and the information that a C-128 was now their in-house system. (Other surprises, currently under development and specifically for the C-128, were alluded to - I will keep you informed.)

CoNIX: The Search

The search for CoNIX will not lead you far. In one of the most savvy marketing moves I recently have come across, Computer Helper Industries releases the previous version of the CoNIX operating system as shareware. This is not a crippled version - it is the full implementation of the previous generation of the software. You are free to use the shareware package for a period of up to six months; at that time you are asked to become a registered user or to destroy the package.

(This was my introduction to CoNIX - downloaded from the CP/M library of the CBMPRG forum on CIS. It took only two months to convince me to place my order.)

The cost of the complete, most recent version of the CoNIX package, including media and shipping via air mail to Canada, was $83.95 US. Various combinations of the CoNIX packages are available for less. Delivery was prompt, under four weeks, a rarity when ordering by mail. The product was well packaged and suffered no damage in transit.

More information on the CoNIX environment can be obtained from: Computer Helper Industries Inc., PO Box 680, Parkchester Station, Bronx, NY, 10462, (212) 652-1786.

You Win Some, You Lose Some

Not everything is perfect with CoNIX. Most annoying is the loss of even the limited command line editing provided with the CCP of CP/M Plus. A recall-last-command command is available but cursor movement commands are limited to a destructive backspace - barbaric!

A more serious problem is the overhead involved in CoNIX use. The many support files, option files, and temporary files need lots of disk storage space. If you are using a single drive system, CoNIX may be your best reason to buy a second storage device. A high-speed, high-capacity storage device like the 1581 3.5 inch disk drive or the 1571 RAM Expansion (used as a RAM disk) is recommended. If you aren't ready to expand your system to this extent, CoNIX isn't for you.

Two features of CoNIX, Expandisk and BDOS patching, must be disabled during installation in order for it to run on the C-128. The procedure for disabling them is clearly explained in the interactive installation program. Without going into their functions, I will say that I have not noticed them in their absence.

I must admit that the number of system crashes has increased since I have begun to use CoNIX. This is to be expected with the opportunities CoNIX provides for the uninitiated to ride roughshod over their systems. Fortunately, the C-128's non-volatile RAM disk reduces the damage a crash can do and makes reboots fast and easy.

Its faults notwithstanding, I would recommend the CoNIX Operating System to anyone who uses the CP/M side of the C-128 with any regularity. If you are using CP/M as your business system, or programming for personal or commercial interests, you will wonder how you got by without the complete CoNIX package.

The redundancy of some CoNIX capabilities when run on the C-128, and the superiority of those C-128 capabilities (redefinable keys, function keys, virtual drives, command line editing), illustrate the power of CP/M Plus on the C-128. However, the addition of CoNIX to this team makes for a truly unbeatable combination.

Great Assignment!

Easy in-program expression evaluation for the C64 and C128

by Paul Durrant

Enter and run the following program on your C64 or C128:

```
10 input"Enter an arithmetic expression";a$
20 a = a$
30 print a
```

At the prompt, enter something like '(13+2)*46', or 'sqr(4)' (without the quotes).

You get a '?type mismatch error', right? I developed a series of inventory control programs which required entering thousands of numbers. Sometimes they were things like: ''13 dozen plus 5 plus eight-and-a-half more dozen.'' Not being able to enter those numbers as an arithmetic expression resulted in considerable frustration - and ''great assignment''. After enabling this routine (by changing the ERROR vector...more later), an arithmetic expression in a string variable can be solved and assigned to a floating point variable. The method of use couldn't be simpler: just execute 'a = a$' (or 'item(x) = entryline$(7)', or...). If the string variable contains a legal arithmetic expression, it will be solved, and its value assigned to the numeric variable. An empty string will be assigned a value of zero. And, the routine stays active, even after doing a 'Run-Stop Restore'.

How it works

Normally, trying to assign the value of a string variable to a numeric variable results in a '?type mismatch error'. In addition, the numeric variable is on the left side of the mismatched equation. So, the new error routine starts by checking for that condition (a '?type mismatch', numeric on the left). If this is not the current error, then it jumps to the normal error handling procedure. If this is the current error, then some additional information is available. The error was recognized after finding the addresses of both variables (numeric on left, string on right). ''great assignment'' uses this information to move the text string into the BASIC input buffer (BBUFF) where CRUNCH can convert the text into executable, tokenized form. Then it calls 'formula.evaluate' to solve the crunched expression and put the result in Floating Point Accumulator #1 (FAC #1). Finally, the assignment statement is completed, using the address of the numeric variable which has been waiting patiently ever since the original error condition.

On the C64, the amount of additional housekeeping required to make this work is minimal. On the C128, things aren't quite so simple. Most of the difference revolves around the issue of where to place the routine. Let's start with the C64, first. There are no internal jumps or subroutines in ''great assignment'', so it can be easily located anywhere in memory. On the C64, the tape buffer will work. So will that ever popular area starting at $C000, and it can even be placed in the BASIC program memory area (if proper adjustments are made to keep BASIC and variables from over-writing it). The program cannot be located ''under'' BASIC or the Kernal, however, since it uses routines contained therein.

On the C128, principles are the same, but location is more complicated. The C128's many memory configurations include several that are used heavily by the BASIC interpreter. Moreover, because the new error handling routine must handle all errors, it must be robust enough to take them on, no matter what memory configuration exists when the error occurs. Only the Common RAM (from $02 to $3FF) can do it, and there isn't room there for even a relatively short program such as this. The solution involves using six free bytes near the end of Common RAM ($3E4 to $3E9). That's exactly enough room to set the desired memory configuration (RAM 0, BASIC and Kernal) and then jump to the remainder of the new error handling routine (which I've placed in the cassette buffer, from $B00). There's more trouble ahead, though: 'crunch', 'frmeval', and the two routines that save and restore the 'txtptr' require RAM 0, BASIC and the Kernal, but MOVE$ uses routines which leave us in RAM 1, BASIC and Kernal. Fortunately, those routines are located in RAM, so ''great assignment'' can change those routines a bit before doing MOVE$, and then restore them to their normal condition when done. The program listing shows it all.

How to use it

To enable ''great assignment'', you must change the ERROR vector to point to the new routine. The ERROR vector is in locations 768 and 769 ($300/301) on both machines. If you place ''great assignment'' at $C000 in the C64, then 'poke 768,0: poke 769,192'. If you use the cassette buffer then 'poke 768,60: poke 769,3'. For the C128 - remembering the six bytes in Common RAM - 'poke 768,228: poke 769,3'. (Or use the Monitor to set $300 to $E4 and $301 to $03.) The BASIC loader programs listed will do everything for you: just run the 64 or 128 version, and start your great assignments!

Annotated Monitor Listing for C-64 "great assignment"

```
c000  b0 08     bcs  $c00a  ;Check for
c002  e0 16     cpx  #$16   ; '?type mismatch error',
c004  d0 04     bne  $c00a  ; with FLPT on left,
c006  24 0d     bit  $0d    ; string on right.
c008  30 03     bmi  $c00d
c00a  4c 8b e3  jmp  $e38b  ;Normal ERROR routine.
c00d  a9 00     lda  #$00   ;Prime to MOVE$ contents
c00f  85 35     sta  $35    ; of $ var to begin
c011  a9 02     lda  #$02   ; of BBUFF.
c013  85 36     sta  $36
c015  a5 64     lda  $64    ;LET erred here with $64/65
c017  85 6f     sta  $6f    ; pointing to header of $ var.
c019  a5 65     lda  $65
c01b  85 70     sta  $70
c01d  20 7a b6  jsr  $b67a  ;MOVE$.
c020  aa        tax         ;= end of $ entered in bbuff+1.
c021  b1 6f     lda  ($6f),y ;(y) = 0 here.
c023  d0 06     bne  $c02b  ;Branch if not a null string.
c025  a9 30     lda  #$30   ;Else enter ASCII "0".
c027  9d 00 02  sta  $0200,x
c02a  e8        inx
c02b  98        tya
c02c  9d 00 02  sta  $0200,x
c02f  a5 7a     lda  $7a    ;Save
c031  a4 7b     ldy  $7b    ; TXTPTR.
c033  85 3d     sta  $3d
c035  84 3e     sty  $3e
c037  a9 00     lda  #$00   ;Set TXTPTR
c039  a0 02     ldy  #$02   ; to BBUFF.
c03b  85 7a     sta  $7a
c03d  84 7b     sty  $7b
c03f  20 79 a5  jsr  $a579  ;CRUNCH.
c042  20 73 00  jsr  $0073  ;chrget (set  txtptr to bbuff).
c045  20 9e ad  jsr  $ad9e  ;FRMEVAL.
c048  68        pla         ;Clear stack
c049  68        pla         ; from
c04a  68        pla         ; TYPE MISMATCH.
c04b  20 72 ab  jsr  $ab72  ;Restore TXTPTR.
c04e  4c d0 bb  jmp  $bbd0  ;MOVE FAC#1 to var.
```

Annotated Monitor Listing for C-128 "great assignment"

```
003e4 8d 03 ff  sta  $ff03  ;Enable ram 0, Basic, Kernal.
003e7 4c 00 0b  jmp  $0b00  ;Jmp to new ERROR routine.

00b00 e0 16     cpx  #$16   ;Check '?type mismatch error'
00b02 d0 04     bne  $0b08
00b04 24 0f     bit  $0f    ; with numeric on left.
00b06 30 03     bmi  $0b0b  ;Found?
00b08 4c 42 4d  jmp  $4d42  ; No: do normal ERROR.
00b0b a9 1b     lda  #$1b   ; Yes: reset ptr to
00b0d 85 18     sta  $18    ; "temp string stack".
00b0f a9 03     lda  #$03   ; make "RAM 1 fetches"
00b11 8d b4 03  sta  $03b4  ; return to RAM 0
00b14 8d bd 03  sta  $03bd  ; during MOVE$.
00b17 a9 00     lda  #$00   ;Set MOVE$'s destination
00b19 85 37     sta  $37
00b1b a9 02     lda  #$02   ; to BBUFF ($0200).
00b1d 85 38     sta  $38
00b1f a5 66     lda  $66    ;Set up MOVE$'s source:
00b21 85 70     sta  $70    ; LET erred with ptr to
00b23 a5 67     lda  $67    ; string in $66/67.
00b25 85 71     sta  $71
00b27 20 4e 87  jsr  $874e  ;MOVE string into BBUFF.
00b2a aa        tax         ;(x) holds length of string.
00b2b d0 06     bne  $0b2f
00b2d a9 30     lda  #$30   ;Place ASCII "0" in BBUFF
00b2f 9d 00 02  sta  $0200,x ; if empty string.
00b32 e8        inx
00b33 98        tya         ;(y) = 0 after MOVE.
00b34 9d 00 02  sta  $0200,x ;End BBUFF with null byte.
00b37 20 34 4b  jsr  $4b34  ;Save BASIC's TEXTPTR.
00b3a a9ff      lda  #$ff   ;Now set
00b3c a001      ldy  #$01   ; TEXTPTR to
00b3e 85 3d     sta  $3d    ; BBUFF - 1.
00b40 84 3e     sty  $3e
00b42 20 0a 43  jsr  $430a  ;CRUNCH the string.
00b45 20 80 03  jsr  $0380  ;Use CHRGET to align ptrs.
00b48 20 ef77   jsr  $77ef  ;'frmeval' solves expression
                            ; (result in FAC #1).
00b4b 20 79 57  jsr  $5779  ;Restore TEXTPTR.
00b4e 68        pla         ;Clean the
00b4f 68        pla         ; stack from the
00b50 68        pla         ; MISMATCH ERROR,
00b51 a9 04     lda  #$04   ;and
00b53 8d b4 03  sta  $03b4  ; restore "RAM 1 fetches"
00b56 8d bd 03  sta  $03bd  ; to normal operation.
00b59 4c fa 53  jmp  $53fa  ;Do assignment; carry on.
```

BASIC loader for the C64 version of "Great Assignment"

```
FB  100 rem "Great Assignment" for the C64
GG  110 for x=49152 to 49232: read a
KO  120 poke x,a: check=check+a: next
AB  130 data 176,   8, 224,  22, 208,   4,  36,  13
OG  140 data 048,   3,  76, 139, 227, 169,   0, 133
LO  150 data 053, 169,   2, 133,  54, 165, 100, 133
AH  160 data 111, 165, 101, 133, 112,  32, 122, 182
EE  170 data 170, 177, 111, 208,   6, 169,  48, 157
BJ  180 data 000,   2, 232, 152, 157,   0,   2, 165
LO  190 data 122, 164, 123, 133,  61, 132,  62, 169
MO  200 data 000, 160,   2, 133, 122, 132, 123,  32
JA  210 data 121, 165,  32, 115,   0,  32, 158, 173
OO  220 data 104, 104, 104,  32, 114, 171,  76, 208
GJ  230 data 187
CJ  240 if check<>8575 then print"You goofed!":end
KL  250 poke768,0: poke769,192: print"OK!"
```

BASIC loader for the C128 version

```
GD  100 rem "Great Assignment" for the C128
KN  110 for x=dec("3e4") to dec("3e9"): reada$
KJ  120 pokex,dec(a$):check=check+dec(a$): next
IH  130 data 8d, 03, ff, 4c, 00, 0b
BC  140 for x=dec("b00") to dec("b5b"): reada$
IL  150 pokex,dec(a$):check=check+dec(a$): next
BD  160 data e0, 16, d0, 04, 24, 0f, 30, 03, 4c, 42
IN  170 data 4d, a9, 1b, 85, 18, a9, 03, 8d, b4, 03
FN  180 data 8d, bd, 03, a9, 00, 85, 37, a9, 02, 85
NE  190 data 38, a5, 66, 85, 70, a5, 67, 85, 71, 20
IM  200 data 4e, 87, aa, d0, 06, a9, 30, 9d, 00, 02
MO  210 data e8, 98, 9d, 00, 02, 20, 34, 4b, a9, ff
CE  220 data a0, 01, 85, 3d, 84, 3e, 20, 0a, 43, 20
EJ  230 data 80, 03, 20, ef, 77, 20, 79, 57, 68, 68
FC  240 data 68, a9, 04, 8d, b4, 03, 8d, bd, 03, 4c
FA  250 data fa, 53
OM  260 data e4, 03
LI  270 read lo$: check=check+dec(lo$)
BG  280 read hi$: check=check+dec(hi$)
NL  290 if check <> 9554 then print"You goofed!":end
HO  300 poke768,dec(lo$): poke769,dec(hi$)
NH  310 print"OK!"
```

Give Me A BRK!

Invisible subroutines on the C64 and C128

by Tom Hughes

I don't think I'd be exaggerating in saying that BRK is the least used 65xx instruction. In fact, BRK is usually associated with disaster - your machine language program wanders off course, slams into a BRK, and your computer ends up in never-never land.

It doesn't have to be this way. BRK can be used to call subroutines that will be "invisible" to all the 65xx registers except the program counter (PC). In a way, BRK can be used as a 6502 equivalent of the 68000's TRAP instruction.

BRKing on the C64 and C128

First, let's see how the Commodore 64 and 128 react to BRKs. When a BRK occurs in the C64 or C128, the PC is loaded with the vector at $FFFE/FFFF, which also serves as the IRQ vector. Since this vector serves a dual function, the computer first must determine what sort of interrupt occurred - an IRQ or a BRK. (Yep, BRK is an interrupt.) The BRK entries for both machines are listed below:

C64:

```
ff48  pha              ;** c64 brk/irq entry **
ff49  txa              ;save .A, .X and .Y on stack
ff4a  pha
ff4b  tya
ff4c  pha
ff4d  tsx              ;current stack pointer to .X
ff4e  lda  $0104,x     ;use it to load old status register
ff51  and  #%00010000  ;test the brk bit in the SR
ff53  beq  $ff58
ff55  jmp  ($0316)     ;if this bit = 1, then this is a BRK
ff58  jmp  ($0314)     ;otherwise, an IRQ.
```

C128:

```
ff17  pha
ff18  txa              ;save .A, .X  and .Y registers
ff19  pha
ff1a  tya
ff1b  pha
ff1c  lda  $ff00       ;also save  current bank on stack
ff1f  pha
ff20  lda  #$00        ;and force bank 15
ff22  sta  $ff00
ff25  tsx              ;current stack pointer to .X
ff26  lda  $0105,x     ;use it to load old status register
ff29  and  #%00010000  ;test the brk bit in the SR
ff2b  beq  $ff30
ff2d  jmp  ($0316)     ;if this bit = 1, then this is a BRK
ff30  jmp  ($0314)     ;otherwise, an IRQ
```

Sifting through the Stack

The BRK entries above are nearly identical for both machines. So why list both? Well, if you're kind of fuzzy on stack operations during an interrupt, then listing both will show you exactly how to get at the .A, .X and .Y registers, the SR, and the PC that have been pushed on the stack - necessary information in order to use BRK effectively.

Notice that the C64 does a 'lda $0104,x', but the C128 uses a 'lda $0105,x'. What's going on? Keep the following in mind:

- the 65XX stack lives between $01FF and $0100.
- when values are pushed on the stack, the stack grows *downwards* in memory.
- the stack is organized in LIFO ("last in, first out") order.

Here's an example of what happens to the stack on the C64 during a BRK, assuming the SP was $F6 when the BRK instruction happened:

```
$01f6        <- old stack pointer
$01f5  PCH   <- program counter high byte
$01f4  PCL   <- program counter low byte
$01f3  SR    <- processor status register
$01f2  .A    <- data registers
$01f1  .X
$01f0  .Y
$01ef        <- SP (= $ef after the entry routine)
```

The C128 stack would look like this (again assuming the SP was at $F6):

```
$01f6           <- old Stack pointer
$01f5   PCH     <- Program counter high byte
$01f4   PCL     <- Program counter low byte
$01f3   SR      <- Processor status register
$01f2   .A      <- Data registers
$01f1   .X
$01f0   .Y
$01ef   BANK
$01ee           <- SP is $EE after the entry routine
```

Note: The program counter (PCH and PCL) and the status register (SR) were saved by the CPU itself when the BRK or IRQ occurred.

Remember that both the entry routines must determine what really happened - a BRK or an IRQ. So both must test the SR saved on the stack. On a C64 the 'lda $0104,x' fetches the old SR from the stack. Since the SP at the end of our entry routine is $EF, just add 4 to this, and 'lda $0104,x' really becomes 'lda $01F3' - the SR. The C128 uses 'lda $0105,x' because it must add 5 to get past the saved bank value. Anyway, use the same technique to get at the other pushed values and where .X is the SP:

For the C64, to find...		For the C128, to find...	
PCH	use LDA $0106,X	PCH	use LDA $0107,X
PCL	use LDA $0105,X	PCL	use LDA $0106,X
SR	use LDA $0104,X	SR	use LDA $0105,X
.A	use LDA $0103,X	.A	use LDA $0104,X
.X	use LDA $0102,X	.X	use LDA $0103,X
.Y	use LDA $0101,X	.Y	use LDA $0102,X
		BANK	use LDA $0101,X

Once these pushed values are located, they can be changed - one method of passing parameters through BRK.

A word on PCH and PCL: the program counter on the stack has had 2 added to it. This is very important to keep in mind while using BRK. The following example shows what happens to the PC afer a BRK:

```
$1000 lda   #$0d
$1002 jsr   $ffd2
$1004 brk               ;($1007 saved as PC on stack)
$1006 nop
$1007 jsr   $ffe4       ;we land here after BRK
```

From the example above, you can see that the PC skips right over the NOP instruction. So it could be any value. In fact, instead of the NOP, we could place a value that our BRK routine could use as some sort of parameter - like a subroutine number. For instance, the code below could be used to call subroutine #5:

```
$1000 lda   #$0d
$1002 jsr   $ffd2
$1005 brk
$1006 .byte #$05        ;subroutine or "trap" #
$1007 jsr   $ffe4
```

The sample source code listed after the article does just that - uses a value after BRK to execute a particular routine or trap.

68000 Traps

Before presenting a sample BRK routine, it might be interesting to look at the trap functions of the 68000 machines, such as the Amiga and the Macintosh.

The 68000 has an instruction called TRAP that allows a programmer to create up to 16 routines that can executed from within a program that generate exceptions or interrupts. (The 68000 also allows something called A-traps and F-traps that are in a sense closer to our use of BRK, but this is beyond the scope of this article.)

Traps allow you to interrupt the microprocessor from software - handy if you need something done in a hurry while at the same time preserving important program values, and the trap will seem invisible to the execution of your program.

Our BRK routine will be invisible because the .A, .X, .Y registers, the SP and, in the case of the C128, the bank value will be unaffected.

A BRK Demonstration Program

The following PAL assembler source code for the C64 consists of three parts:

(1) A routine to divert the standard BRK vector at $0316/0317 to our custom routine.

(2) A BRK handler that shows how to incorporate BRK in a machine code program. One small note: this part of my program contains self-modifying code because TRPNM is changed each time you type 1, 2 or 3. This is not good programming practice (try using the code in an EPROM!) and was only done to shorten the example. If you wish to make use of the concept described in the article, I suggest fixed trap numbers following the BRK instruction.

(3) The BRK routine itself simply prints 1, 2 or 3 to the screen depending on which trap was used. The TIDYUP routine pulls the registers from the stack and does an RTI.

Final Notes

Interrupt priority: traditionally, only the NMI and IRQ are mentioned during any discussion of interrupts. Actually, the interrupt priority, from highest to lowest, is as follows: NMI, BRK, IRQ. A BRK supersedes an IRQ because IRQs are disabled by the SEI instruction, but not BRKs.

BRK to BRK? Though I haven't tried it, I suppose you could write a program using BRKs instead of JSRs. However, watch nesting BRKs inside of BRKs because each will need 6 (or 7 for a C128) slots on the stack, and there's only 256 stack locations to work with before the stack wraps around and obliterates some vital parameter.

Keyboard scanning: since the Kernal's IRQ routine is responsible for scanning the keyboard, you will have to use the SCNKEY Kernal routine if you want keyboard input to be read while inside a BRK. Here's how to read and print a character in this case:

```
loop    jsr  scnkey     ;scan keyboard
        jsr  getin      ;read it
        beq  loop
        jsr  chrout     ;print it
```

PAL-format source code to demonstrate simulating a trap with the BRK instruction:

```
GD  10 sys 700: .opt oo
OB  1000 ;*********************************
MB  1010 ;*                               *
MH  1020 ;*  simulating a trap with brk    *
DE  1030 ;*  -------------------------     *
KD  1040 ;*                               *
EE  1050 ;*                               *
HN  1060 ;* brk vector is diverted so     *
IL  1070 ;* that "invisible" subroutines  *
FF  1080 ;* can be called.                *
MG  1090 ;*                               *
GH  1100 ;*                               *
BM  1110 ;*  - by tom hughes  v022287 -    *
KI  1120 ;*                               *
AK  1130 ;*********************************
KO  1140 ;
OJ  1150 ;c64 equates
OP  1160 ;
JJ  1170 cbinv   = $0316 ;brk vector (2)
CL  1180 chrout  = $ffd2  ;output a char
PN  1190 clrchn  = $ffcc  ;i/o to defaults
AF  1200 getin   = $ffe4  ;input a char
BD  1210 memory  = $8d    ;temp storage (2)
EK  1220 oldbrk  = $8b    ;storage for standard brk (2)
IO  1230 stack   = $0100 ;65xx stack location
BI  1240 tidyup  = $febc  ;recover from interrupt
LP  1250 *= $c000         ;sys 49152
BN  1260 ;------------------------------
CK  1270 ;set brk vector to our routine
FO  1280 ;------------------------------
HJ  1290 ;in actual use, this would be a subroutine
EG  1300 ;called once to divert the brk vector.
EJ  1310 ;
PC  1320        jsr  clrchn
MC  1330        sei              ;disable interrupts
AD  1340        ldx  cbinv
FC  1350        ldy  cbinv+1
AD  1360        stx  oldbrk      ;save old brk vector
FM  1370        sty  oldbrk+1
AM  1380        ldx  #<newbrk    ;then set new vector
BJ  1390        ldy  #>newbrk
KK  1400        stx  cbinv
PJ  1410        sty  cbinv+1
OG  1420        cli              ;enable interrupts
MA  1430 ;
FI  1440 ;------------------------------
KP  1450 ;demo brk handler
JJ  1460 ;------------------------------
ED  1470 ;
CC  1480 ;this is just an example of how you
AD  1490 ;would use brk from within a program
CF  1500 ;
NE  1510 demo5   ldy  #0
HN  1520 demo10  lda  prompt,y   ;print "number?"
EB  1530        beq  demo20
LN  1540        jsr  chrout
OO  1550        iny
MB  1560        bne  demo10
OC  1570 demo20  jsr  getin      ;check the keyboard
HL  1580        cmp  #3          ;(if stop key, quit)
DP  1590        beq  quit
ID  1600        cmp  #"1"        ;for numbers 1 thru 3
GC  1610        bcc  demo20
MI  1620        cmp  #"4"
KH  1630        bcs  demo20
PD  1640        jsr  chrout
LE  1650        and  #$0F        ;make # hex 1 - 3
EA  1660        sta  trpnm       ;save in our own prog
NC  1670        lda  #13         ;print a carriage gosub
HG  1680        jsr  chrout
JB  1690        brk              ;execute trap
EF  1700 trpnm   .byt 0           ;(trap #)
HL  1710        jmp  demo5
          ;(after brk, prg continues here)
OC  1720 ;
EH  1730 quit    sei
DD  1740        ldx  oldbrk
DA  1750        ldy  oldbrk+1
CB  1760        stx  cbinv
HA  1770        sty  cbinv+1
CI  1780        cli
DI  1790        rts              ;back to basic
OH  1800 ;
DC  1810 prompt  .byt 13,13
KI  1820        .asc "number (1 - 3)? "
AG  1830        .byt 0
AD  1840 ;==============================
NA  1850 ;new brk routine
EE  1860 ;==============================
EM  1870 ;
CI  1880 ;entry (1) interrupts disabled (except nmi)
FO  1890 ;         so jiffy clock is off.
CO  1900 ;
HG  1910 ;(2) on entry stack looks like this...
GO  1920 ;   (assuming old sp was at $f6)
```

```
AA  1930 ;
BJ  1940 ;       $01f6          <- old sp
PM  1950 ;       $01f5 pch      (stack+6)
MN  1960 ;       $01f4 pcl      (stack+5)
HL  1970 ;       $01f3  sr      (stack+4)
IB  1980 ;       $01f2  .a      (stack+3)
FD  1990 ;       $01f1  .x      (stack+2)
MD  2000 ;       $01f0  .y      (stack+1)
GF  2010 ;       $01ef          <- current sp
KF  2020 ;
OH  2030 ;(3) expects trap # after brk
JP  2040 ; (this location can be found by
FE  2050 ; using the pc saved on stack -1.)
CI  2060 ;
DK  2070 newbrk  tsx                    ;get current sp to .x
CH  2080         lda  stack+6,x         ;to find pc-high
OI  2090         sta  memory+1
CJ  2100         lda  stack+5,x         ;and pc-low on stack
DO  2110         sta  memory            ;save this address
BB  2120         bne  new10             ;and subtract -1 from it
NK  2130         dec  memory+1          ;so we can locate trap#
BB  2140 new10   dec  memory
BL  2150         ldy  #0
MH  2160         lda  (memory),y;get trap #
BN  2170         tay                    ;adjust it so 1-3
IK  2180         dey                    ;is now 0-2
FE  2190         tya
LH  2200         asl  a                 ;multiply this # by 2
BH  2210         tay
JF  2220         lda  table,y           ;and use it to look up
KH  2230         sta  memory            ;trap addresses
AK  2240         iny
GK  2250         lda  table,y
ID  2260         sta  memory+1
KE  2270         jmp  (memory)          ;go to a trap routine
OF  2280 ;
JD  2290 ;trap addresses
CH  2300 ;
FC  2310 table   .word trap1
LM  2320         .word trap2
HN  2330         .word trap3
KJ  2340 ;
NK  2350 ;//////////////////////////////
LM  2360 ;demo trap routines
BM  2370 ;//////////////////////////////
CM  2380 ;
BH  2390 trap1   lda  #"1"
HD  2400         jsr  chrout
MB  2410         jmp  tidyup            ;must end with this
KO  2420 ;
PJ  2430 trap2   lda  #"2"
PF  2440         jsr  chrout
IF  2450         jmp  tidyup
CB  2460 ;
NM  2470 trap3   lda  #"3"
HI  2480         jsr  chrout
AI  2490         jmp  tidyup
AK  2500 .end
```

Micro-Lisp Version 2.5

A public domain Lisp interpreter for the C64!

by Nicholas Vrtis

Lisp is a language designed to work with lists (its name is a contraction of LISt Processor). It is one of the primary languages used in the study of Artificial Intelligence. Micro-Lisp is a subset of this language that you can use to learn more about its capabilities. Although there obviously isn't space in this article for a complete course in Lisp and AI, I would like to introduce you to the language and to my Micro-Lisp implementation in particular. At the end of the article you'll find some suggestions for further reading if you want to know more.

Why bother with a version of Lisp that runs on a slow 8 bit computer? Because it is an easy, inexpensive way to become familiar with the language, and to get a feel for what it is like. Why buy a model rocket? It can't go as fast or as far as the Space Shuttle, but you can still have fun and learn from it.

Lisp's World View

Lisp divides the world into two classes of 'things'. On one side there is a *List*, and on the other is things that aren't Lists. Things that aren't Lists are called *Atoms*. As in physics, an Atom can't be broken down into something smaller (though you will find that you can *explode* and *implode* an Atom, just as in physics atoms can be taken apart if you know how). A word is an Atom, so is a number. If I take some Atoms, and collect them, I end up with a List (similar to taking atoms and collecting them into molecules). I can also take Lists and group them together, either end-to-end to make a longer List, or as a List of Lists.

Big deal - what good are lists anyway? Actually, if you think about it, almost all the information we use is in the form of lists. Your checkbook for example, is a list of three or four items of information about each check (the check number, the amount, who it is to, the date written, and possibly a budget category). To get the amount I have spent on a given category, I just go through my list of checks and add up the amounts for those checks with that category. That's a relatively simple Lisp application, and maybe not a very appropriate one - a good database program would probably be better, since it would allow you to sort the data and print a fancy report without a lot of work.

Lisp was designed to handle more complicated situations where you can't know in advance all the combinations and questions you might want to ask about the information you have. One example (which I'll be using throughout this article) would be a family tree. A List showing the name, sex, and parents of each individual in your family would be a good starting point. Each item on this List is made up of two Atoms (name and sex) and one List (Father's name, Mother's name). Notice that there is no item in the List concerning the individual's relationship to you, or to most other members of the family. We can get this information, however, by applying some simple rules; for instance: *a brother of x is any male whose parents are the same as the parents of x.* Before we discuss how Lisp lets us extract this kind of implicit information, however, we need to master a few of the language's technicalities.

Lisp Fundamentals

Lists in Lisp are enclosed in parentheses. For instance, while *nick* is an Atom, *(nick)* is a List that has one Atom, *(nick m)* has two, and *(nick m (jim marion))* has two Atoms and a List (which itself has two Atoms). By the way, one of the hard parts about Lisp, especially for beginners, is keeping the parentheses balanced in the right places. Micro-Lisp has a couple of features to help with this. The command prompt shows the current number of unbalanced parentheses (the *nesting* level). Also, when you display a List, you can use a feature called *pretty print* to start each new level on a new line, and indent one space for each level.

Another concept you need to understand about Lisp is how it represents 'nothing'. Since an Atom can be either a number or a word, Lisp can't use 0 for numbers the way we do. Instead, Lisp uses an entity called *nil*. *Nil* is special, because it can be either a List or an Atom depending on the situation. If you want to input the Atom *nil*, just enter the word *nil*. If you want to input a List with nothing in it, enter *()*. Whenever Micro-Lisp displays an empty List, it will always display *nil* instead of *()*.

We also need to understand how to get Lisp to do something - how to give it a command. Commands are given in the form of Lists (not surprisingly). A command List is no different from any other List, except that the first entry must be an Atom that is a command. For example, *add* is a command that sums the numbers in the rest of the List; the List *(add 1 2 3)* would add the numbers 1, 2 and 3. A List doesn't have to have a command as the first entry unless you want to execute it (called *evaluating* it in Lisp). The documentation accompanying this article shows the built-in commands available in Micro-Lisp. One of these - *define* - lets you create your own commands, which work just like the built-in commands. We'll make use of this ability when we work on our family-tree project.

Creating the family tree

Let's begin that now. To start Micro-Lisp, just enter:

```
load "micro-lisp",8
run
```

You'll see a title message and a flashing cursor. Now type:

```
>0 (set (quote family-tree) nil)
nil
```

The computer will respond by typing out *nil* (note: the examples in this article use **bold** type for the computer's prompts and responses, and regular type for your input). *Set* is a command that sets the value of the second Atom in the List equal to the value of the third (similar to a BASIC statement like FT$ = ""). Now we have a 'database' named FAMILY-TREE with nothing in it.

You might be wondering why you had to use the strange construction *(quote family-tree)* instead of just *family-tree*. This reflects LISP's desire to use the *value* of a name in most cases. Consider the BASIC statement FT$ = A$, which assigns the value of A$ to FT$. If we really wanted to assign the characters "A$" to FT$, we have to use quotes. The *quote* command in LISP performs the same function as the pair of double quotes in BASIC. If that is still confusing, try this:

```
0> (set (quote tree-name) (quote family-tree))
family-tree
0> (set tree-name 10)
10
0> tree-name
family-tree
0> family-tree
10
```

In this example, we begin by creating a new Atom - *tree-name* - whose value is the name *family-tree*. When we now say *(set tree-name 10)*, we are asking for the value 10 to be assigned to the Atom whose name is found by evaluating *tree-name*. After this operation, we discover that the value of *tree-name* is unchanged but, as expected, *family-tree* has the new value 10. Programmers who have used languages like assembler, C and PROMAL will recognize here an example of *indirection*; this application of it is fundamental to Lisp and you should make sure you understand the above example thoroughly.

Since you end up using the *quote* command a lot in Lisp, there is a shorthand version. It is a single quote mark ('). It eliminates the word *quote* and a set of parentheses. We can thus write our original statement more concisely as:

```
(set 'family-tree nil)
```

Go ahead and try it - Lisp is very interactive. If you ever want to know the value of a name, just type it on the command line without parentheses.

A new command with *define*

Now let's define a command of our own to add a person to our database. We use the *define* command for this, and we'll keep it simple for now. Later you will probably want to add some checks to this command to guard against duplicate entries and to determine if the parents of a newly-added person are already defined. But start with:

```
0> (define 'add-person '(person)
1>    '(progn
3>          (setq family-tree
4>             (cons person family-tree))
3>          person))
```

Micro-Lisp should respond with *add-person*. If not, make sure you haven't missed any quotes or parentheses (hint: if you have a number greater than zero in front of the > prompt, that is the number of parentheses you are missing). You don't have to indent as shown above, though it helps show each level of the definition.

There are a number of new items in this definition, but most are pretty simple. *Define* is the command that defines new commands to Lisp. It needs to know three things. The first is the name of the command (we are calling it *add-person*); the second is a List of the arguments (only one in this case - *person*); and the last is the body of the function. Note that quote marks are used, since we want the literal statements we typed in, not their value.

The first command in the function body is *progn*. All this command does is tell Lisp to evaluate all the other items in the List. Normally, Lisp expects a command to be in the form (Command Argument Argument...). *Progn* lets you string a set of commands into one List in the form *(progn (Command Argument ...) (Command Argument ...) ...)*. We need to do this in *add-person* because we want to do two things. The first is *setq*. This is a special version of *set*, which we used earlier. *Setq* allows you to skip the first quote. The variable we are

setting is *family-tree*, our database. What we want to set it to is a List consisting of all the things already in *family-tree* plus the new *person* data. We use the *cons* command to do this. *Cons* creates a List formed from the first argument followed by the second argument. Note that I put *person* first, and *family-tree* second, thus adding the new information to the front of the database instead of the end. For technical reasons this turns out to be faster than the other way around, but either way will work.

The second item is not a command, just the word *person*. Notice that it is not enclosed in parentheses. When Lisp sees just a variable name without parentheses, it just takes the value of that variable, and leaves it as the *return value*. Everything in Lisp leaves some sort of return value; whenever Lisp has finished processing the commands you have given it, it prints out the final return value. Well, it turns out the return value from *setq* is the value to which the variable was set. In our case, this is the whole database. Since it could get lengthy to have it print out every time we add a new person to *family-tree*, we add the word *person* by itself; now our new command has as its return value the information about the person we just added. We could have used a special variable called *t*, or *nil*, but *person* might be more useful if we want to combine *add-person* into some other command.

Let's test out what we have so far. Issue the following:

```
0>(add-person '(nick m (jim marion)))
(nick m (jim marion))
0>family-tree
((nick m (jim marion)))
```

The last should produce a List of Lists showing everything in our database. Since it is all scrunched together, enter *(setpretty t)*, then *family-tree* again. This will indent each level of parentheses and make the Lists a little easier to read. Now build up the database a little by adding some more people with the following lines (this time, Lisp's responses are not shown):

```
(add-person '(maryelna f (frank dorothy)))
(add-person '(nikki f (nick maryelna)))
(add-person '(mike m (jim marion)))
(add-person '(matt m (nick maryelna)))
family-tree
```

Notice that the database has become larger.

Interrogating the database

Now let's define some new commands that will help us find things in the database. The first is a command to find somebody's name and parentage:

```
0>(defun find-name (name)
1>    (setq temp family-tree)
1>    (dountil (or
3>          (eq temp nil)
3>          (eq name (car (car temp))))
2>       (setq temp (cdr temp)))
1>    (setq person (car temp)))
```

There are more new commands here, but it is still pretty simple. *Defun* is another version of *define*. Like *setq*, it is a shorthand that eliminates the need for quoting its arguments. *Defun* also supplies an implicit *progn*, so we don't have to bother with that either. We've seen *setq* before. Here we are using it to set a temporary variable to our database because we are going to have to check each item in it to see if the first Atom of the sublist is the name we are looking for.

Dountil is a looping command. Until the first argument returns a true value (in Lisp, *true* means anything that isn't *nil*), this command will execute the remaining commands in the List. In our case, we will want to terminate the loop when either we have run out of person entries in the database, or we have found the person entry whose first Atom is the name of the person we are looking for.

Conveniently, Lisp has an *or* command to express this sort of requirement. *Or* evaluates each of its arguments until it either finds a non-*nil*, or runs out of arguments (in which case it returns *nil*). The first argument is *(eq temp nil)*. The *eq* tests to see if its two arguments are identical, so if *temp* is equal to *nil*, this command will return *t*. If *temp* is not *nil*, *or* goes to the next argument. This means there is a person List left, so we want to compare the first Atom in that List with the name we are looking for.

To do this, we use the *car* command. *Car* returns the first part of its argument (which must be a List). Since *temp* is a copy of *family-tree*, the *car* (first item) of *temp* is the first 'person' List in *temp*. The name is the first Atom in the List, so we take the *car* of the *car* of *temp*, and compare that to the name we're looking for. If they are the same, the *eq* will return *t*, and the *dountil* is done. Otherwise we need to do something to look at the rest of the person Lists in *temp*.

This is where *cdr* comes in. *Cdr* returns the tail of a List - everything but the *car*. Since *or* told us that the current first List in *temp* isn't the one we want, we simply *setq temp* to everything but the first List, and repeat the process. Eventually, the *dountil* either runs out of Lists in *temp* (*temp* equals *nil*), or the *car* of *temp* is the person *list* corresponding to the name we want. When the *dountil* terminates, we return the *car* of *temp*. Note that the *car* of *()* is *nil*, so *find-name* returns either the person List of the name we asked for, or *nil* if the name is not found. Try *(find-name 'nikki)*; you should get back *(nikki f (nick maryelna))*.

Now for another simple, but useful command:

```
0>(defun parents-of (name)
1>    (cond
2>       ((find-name name) (setq parents (nth 3 person)))
```

```
2>      (t (setq parents nil))))
parents-of
```

Pretty easy, right? Only two new commands this time. *Cond* is the Lisp version of *if*, but a little more complicated. Basically, the arguments for *Cond* are the members of a List of *if* statements. Each argument List is a pair of commands. The first command in the pair is the condition part. If it returns non-*nil*, then the second command is evaluated. If the first command returns *nil*, then the next condition is examined. There is a requirement in Micro-Lisp that at least one of the conditions in a *cond* statement must return non-*nil*, or it is considered an error.

In *parents-of*, the first condition is *(find-name name)*. Recall that *find-name* returns the person List entry if the name is found, or *nil* if it is not. If *(find-name name)* returns non-*nil* in the present case, we will want to set a variable called *parents* to the third item in the person List that *find-name* returned. To do this we use *nth*, a command that returns the nth entry from the third argument (a List), where *n* is specified by the second argument.

In case the first condition returns *nil* (the name was not found), we need to make sure that at least one condition is true (non-*nil*). Lisp supplies a variable called *t* that is guaranteed to return non-*nil*; we use this for the second condition, and set *parents* to *nil* because we can't identify the parents of someone not in the database. Note that Lisp skips condition testing after the first true condition is found, so the second *setq* in the above definition is never executed if the name we are looking for is found. Try *(parents-of 'nick)*; you should get back *(jim marion)*.

One more short example:

```
0>(defun grand-parents-of (name)
1>    (setq grand-parents
2>        (list
3>            (parents-of (car (parents-of name)))
4>                (parents-of (car (cdr (parents-of name)))))))
```

Only one new command in the whole thing, and it is pretty easy to figure out what it does. *List* takes all its arguments and returns a List (simple, isn't it?). Think about what is going on. Grandparents are parents of a person's parents, so all our new command has to do is create a List of the parents of the parents of the person in *name*. *Parents-of* returns a List with the two parents' names. The *car* (first part) of this List is the name of one of the parents. If we now call *parents-of* with this, we will get one set of grandparents. The *cdr* (rest of) the original parent List is a List (*cdr* always returns a List) that has only one entry, the other parent. The *car* of that is the name of the other parent, and the *parents-of* that is the other set of grandparents. It takes a long time to explain, but it really isn't complicated - just follow it through. Try *(grand-parents-of 'matt)*; you should get back *((jim marion) (frank dorothy))*.

Where to go from here

I could continue with more examples, but these should give you an idea of what Lisp is. Purists will probably be upset that I did not use recursion techniques in the examples. Lisp handles these very well, but I find them difficult to follow and harder to explain; I purposely kept the examples straightforward. As you can see by examining the list of built-in commands, there are a lot of Lisp words that I didn't even cover. Experiment with them. Even more than BASIC, Lisp is interactive. Try some things and see what happens.

After you have some experience, try the command called *setdebug*. This turns on a trace facility that traces what is going on. Then use *baktrack* to see all that went on to get to where you are (with our simple *family-tree*, *grand-parents-of* goes through over 100 Lisp statements to get the answer). There is also a *trace* command that prints out the levels as they are being executed. If *setdebug* is *t* and there are symbolic variables (as *name* was in our examples) you get an opportunity to display their values before Lisp restores them (enter *nil* to get out of this mode).

There is an editor (entered with the *edit* command) that allows you to input and save Micro-Lisp source statements; you use the Micro-Lisp command *source* to load them into your Lisp 'environment'. To Micro-Lisp, your 'environment' is all the Lists and Atoms you have defined. Use the commands *save* and *load* to keep and restore copies of your work.

A separate program ("sae.lisp") is a special version of the Micro-Lisp *edit* command. This program runs without Micro-Lisp to let you create Micro-Lisp source programs larger than would be possible with Micro-Lisp running (since Micro-Lisp and all your working Lists take up memory).

Meanwhile, if Lisp interests you enough that you would like to know more about it, you might want to read some or all of these books and articles:

Programmer's Guide To Lisp, by Ken Traction (Tab Books)
Understanding Lisp, by Paul Gloess (Alfred Publishing)
Lisp: Basically Speaking (*80 Micro*, May 1983)
Design Of AN M6800 Lisp Interpreter (*Byte*, August 1979)
Three Microcomputer Lisps (*Byte*, September 1981)

* * * * *

***Editor's Note**: Unfortunately, with a 10K object file size, there is just no room for a program listing of Nick Vrtis' Micro-Lisp interpreter in the magazine. However, Micro-Lisp, some sample Lisp code and the SAE.lisp stand-alone editor will be available on the disk for this issue, and will also be posted to Data Library 17 on CompuServe's CBMPRG Forum. In the near future, **Transactor** will also be releasing a special disk containing the MAE assembler source to Micro-Lisp, along with programming notes for those who wish to expand or modify the interpreter for their own needs.*

Micro-Lisp Built-In Functions

ABORT **(ABORT msg)**

Stops current processing, displays the message that *msg* evaluates to, and returns to the top level processing. This function can be used to define additional error processing.

ABS **(ABS n)**

Returns the absolute value of *n*.

ADD **(ADD n1 ... nn)**

Returns the sum of the numbers *n1* through *nn*.

AND **(AND x1 ... xn)**

Evaluates *x1* and, if it is not *nil*, proceeds to evaluate the following arguments until *nil* is returned, or the end of the argument List is encountered. Note that *and* does not evaluate the following arguments if *nil* is returned from any argument.

APPEND **(APPEND l1 l2)**

Returns the List created by appending *l2* to the end of *l1*.

APURGE **(APURGE atm)**

Purges the Atom *atm* from memory and frees the space for reuse. Any references to *atm* in Lists are changed to reference *nil*. *Apurge* removes the value, definition, and property Lists of built-in functions, but does not remove the built-in definition or free the space.

ATOM **(ATOM x)**

Returns *t* (true) if *x* evaluates to an Atom; returns *nil* otherwise.

BAKTRACK **(BAKTRACK n)**

Displays the last *n* functions that were executed. Only valid if *debug* has been previously activated (see *setdebug*).

CAR **(CAR list)**

Returns the first part of *list*.

CDR **(CDR list)**

Returns the rest of *list* after the first part (the *car*) is skipped.

COLD **(COLD)**

Restarts Micro-Lisp from the beginning. All options and parameters are reset to the way they were when RUN was issued.

COND **(COND (t1 r1)(t2 r2))**

Evaluates *t1* and executes *r1* if *t1* returns non-*nil*. If *t1* returns *nil*, *cond* proceeds to evaluate *t2*. Note that *t2* will not be evaluated if *t1* does not evaluate to *nil*. An error is issued if all tests evaluate to *nil*.

CONS **(CONS x1 x2)**

Returns a List that has *x1* as its first part (*car*) and *x2* as its rest part (*cdr*). If *x1* and *x2* evaluate to Atoms, a special List called a 'Dotted Pair' is returned.

DECIMAL **(DECIMAL)**

Sets the current number base to decimal and returns the number 10.

DEFINE **(DEFINE fn arg x)**

Create a function called *fn* that has arguments specified in the list *arg*. The body of the function is specified in the list *x*. In *define*, *fn*, *arg*, and *x* are evaluated. A defined function has an implied *progn* before the body.

DEFUN **(DEFUN fn arg x)**

Same as *define*, except that *fn*, *arg*, and *x* are not evaluated. It is a more convenient form when entering a function from the keyboard since quotes are not needed.

DISKCMD **(DISKCMD msg)**

Issues the disk command *msg*. This function opens and closes the disk command channel.

DISKST **(DISKST)**

Reads the disk status via the command channel. The function opens and closes the command channel.

DISK$** **(DISK$)

Reads the disk directory and returns a List of Lists about the disk. Each entry is a List of 3 Atoms. The first Atom is the number of blocks, the second is the file name, and the third is the file type. The first entry is the disk name, and the last entry is the number of blocks free.

DIVIDE **(DIVIDE n1 n2)**

Returns the integer result of *n1* divided by *n2*.

DOUNTIL **(DOUNTIL t x1 ... xn)**

A looping function. The test *t* is evaluated and, until it is true (non-*nil*), the expressions *x1* through *xn* are evaluated (using an implied *progn*). *Dountil* returns the non-*nil* value returned from the expression *t*.

DOWHILE **(DOWHILE t x1 ... xn)**

A looping function. The test *t* is evaluated and, while it is still true (non-*nil*), the expressions *x1* through *xn* are evaluated (using an implied *progn*). *Dowhile* returns *nil* always.

EDIT **(EDIT)**

Exits Lisp to the BASIC screen editor. You may enter a Lisp program as you would a BASIC program. The only keywords that will function are LOAD, SAVE, LIST, NEW and END. END returns you back to Micro-Lisp, the others operate as expected.

EQ **(EQ x1 x2)**

Returns *t* if *x1* is identical to *x2*. Not very useful if comparing Lists.

EQUAL **(EQUAL x1 x2)**

Returns *t* if *x1* is the same as *x2*, otherwise returns *nil*. This will properly test Lists.

EVAL **(EVAL x)**

Evaluates the List that *x* evaluates to. *(Eval '(car '(a b)))* will produce the same results as *(car '(a b))*.

EXIT **(EXIT)**

Leaves Lisp and returns to BASIC.

EXPLODE **(EXPLODE a)**

Returns a List of numbers that represent the ASCII value of the name of the Atom *a*. If *a* evaluates to a number, it is converted according to the current base and the ASCII values of the digits are returned.

GC **(GC)**

Forces garbage collection to take place. Returns *t*.

GETDEF **(GETDEF a)**

Returns the definition of the function specified by *a*. Returns *nil* if *a* has not been previously *defined*.

GETPROP **(GETPROP a pn)**

Returns the property value stored under the property name *pn* under the atom *a*. Returns *nil* if *pn* is not defined under *a*.

GREATERP **(GREATERP n1 ... nn)**

Returns *t* if the numbers *n1* through *nn* are in descending order. Note that equal is not considered descending.

HEX **(HEX)**

Set the current number conversion base to 16 (hexadecimal). Returns the value 10 hex (decimal 16).

IMPLODE **(IMPLODE l)**

Takes a List of numbers representing ASCII characters and returns an Atom whose print name is that series of ASCII characters. If any number is greater than 256, the ASCII character is taken from the MOD 256 value. If the print name resulting from imploding the List results in a valid number, then it will be converted to a number.

LAMBDA **((LAMBDA (x) (b)) y)**

A method of executing a function without defining it. The current value of *x* is saved, then it is assigned the value of *y* and the body *b* is evaluated (with an implied *progn*). After *b* is evaluated, *x* is restored to its previous value. Note that there may be more than one Atom specified in the argument list, and that there must be a one to one correspondence between the number of arguments and the number of values supplied.

LENGTH **(LENGTH x)**

Returns the number of elements in the list *x*.

LESSP **(LESSP n1 ... nn)**

Returns *t* if the numbers *n1* through *nn* are in ascending order. Note that equal is not considered ascending.

LIST **(LIST x1 ... xn)**

Returns a List containing *x1* through *xn*.

LISTP **(LISTP x)**

Returns *t* if *x* evaluates to a List, *nil* if *x* evaluates to an Atom.

LOAD **(LOAD fn)**

Loads a previously saved environment from file *fn*. This must be executed from the first level.

LPAREN **(LPAREN)**

Outputs the left parenthesis character.

MEM **(MEM)**

Prints out the number of free object entries, the number of free List entries and the amount of unallocated memory (in bytes). Returns the amount of unallocated memory.

MULTIPLY **(MULTIPLY n1 ... nn)**

Returns the product of *n1* times *n2* ... times *nn*. Note that no check is made for overflow.

NEW **(NEW)**

Clears memory of any user defined Atoms, Lists, and functions without changing settings such as pretty print, echo, or number base.

NIL **NIL, (NIL), or ()**

This is the Lisp specification of 'nothing'. As a value it returns *nil*; as a function it also returns *nil*. It is also considered both an Atom and a List.

NTH **(NTH n l)**

Returns the *n*th element of the List *l*.

NULL **(NULL x)**

Returns *t* if *x* evaluates to *nil*, returns *nil* otherwise (performs a logical NOT function).

NUMBERP **(NUMBERP x)**

Returns *t* if *x* evaluates to a number, *nil* otherwise.

OR **(OR x1 ... xn)**

Evaluates *x1* and if it is *nil*, *or* proceeds to evaluate the following arguments until a non-*nil* value is returned, or the end of the argument list is encountered. Note that *or* does not evaluate the following arguments if non-*nil* is returned from any argument.

PATOM **(PATOM a)**

Prints the Atom *a*.

PRINT **(PRINT x)**

Prints the expression *x* evaluates to, followed by a carriage return. *Print* returns the value *t*. If the pretty print flag is set, each parenthesis level will be started on a new line, and will be indented.

PROGN **(PROGN (x1) ... (xn))**

Successively evaluates the specified function expressions, *x1* through *xn*.

PUTPROP **(PUTPROP a pn pv)**

Puts the property value *pv* as the property *pn* of Atom *a*. If the property *pn* already exists, the old value is replaced by the new value.

QUOTE **(QUOTE x)**

Returns the unevaluated expression *x*.

RATOM **(RATOM)**

Waits for a single Atom to be entered from the terminal.

READ **(READ)**

Reads an expression from the terminal.

RPAREN **(RPAREN)**

Outputs a right parenthesis to the terminal.

SAVE **(SAVE fn)**

Saves the current environment (including all objects and Lists) to file *fn*.

SET **(SET a x)**

Causes the value of *x* to be assigned to the Atom *a*.

SETBASE **(SETBASE n)**

Sets the number base used for conversion for both input and output. Note that *n* is converted and evaluated with the *current* base before the new number base is set.

SETDEBUG **(SETDEBUG x)**

If *x* evaluates to *nil*, debug mode is turned off; if it evaluates to non-*nil*, debug mode is turned on. Debug mode is useful for problem determination. If debug mode is on, Micro-Lisp will track up to the last 128 functions. This tracking can be reviewed using the *baktrack* function. Unlike *trace*, which displays the actual input arguments to a function, *debug* only tracks the unevaluated arguments. Debug mode also gives you an opportunity to print any Atom values before *lambda* and function arguments are restored in error processing. This is sometimes useful in determining what caused an error condition to occur.

SETECHO **(SETECHO x)**

Used to control the echo of *source* input to the terminal. *(Setecho t)* will cause all *source* input to be echoed (this is the default setting). *(Setecho nil)* will eliminate the echo.

SETPRETTY **(SETPRETTY x)**

Used to set the pretty print flag. If *x* is non-*nil*, pretty printing will be turned on and expressions will be printed with each parenthesis on a new line and indented for easier reading. If *x* is *nil*, the flag is turned off.

SETQ **(SETQ a x)**

Causes the value of *x* to be assigned to the Atom *a* (similar to *set*, except that for *setq*, *a* is not evaluated).

SOURCE **(SOURCE fn)**

Directs that input is to come from the disk file *fn* instead of the keyboard. Input is obtained from there until the end of the source file. The source file may contain another *source* statement. This will close the current source file and open the new source file for input.

SUBTRACT **(SUBTRACT n1 n2)**

Returns the value *n1 - n2*.

SYS **(SYS adr x y a f)**

Invokes a machine language routine at address *adr*. The values for the *x*, *y*, *a* and *f* (flag) registers are optional and specified by the corresponding arguments. This function will not allow the IRQ flag to be set, but any other processor flags can be set with the *f* argument. This function returns a List of numbers consisting of the values of the X, Y, A and FLAG registers after the return from the machine code call.

T **T, (T)**

T is one method of specifying a non-*nil* value. As a function, *t* returns *t*.

TERPRI **(TERPRI)**

Causes a carriage return to be output.

TRACE **(TRACE x)**

Causes each function level and its arguments to be output to the terminal as it is evaluated. Note that the actual evaluated arguments are output.

UNDEF **(UNDEF a)**

Removes the function definition from the Atom *a*. If *a* was a native function that had been redefined, the native definition will be restored.

ZEROP **(ZEROP n)**

Returns *t* if *n* is equal to zero; returns *nil* if it is not.

+ **(+ n1 ... nn)**

Shorthand for *add*.

- **(- n1 n2)**

Shorthand for *subtract*.

***** **(* n1 ... nn)**

Shorthand for *multiply*.

(/ n1 n2)

Shorthand for *divide*.

Special Input Characters

^ Used to start and end a comment. All characters between the first ^ and the second ^ are ignored. (Note: this character prints on your C64 screen as an up-arrow.)

' Used as a shorthand for (*quote* ...) - *(quote x)* can be shortened to *'x*. Note the dropping of the word *quote* and a set of parentheses. *'X* will print out as *(quote x)*.

" Used to allow special characters and spaces to be included in an Atom name. Any characters between the double-quotes will become part of the Atom name. The double-quotes themselves will not become part of the name.

$ Used to specify a base 16 (hex) number regardless of the current setting of *base*. Must precede any digits.

. Used to specify a base 10 (decimal) number regardless of the current setting of *base*. Must precede any digits.

% Used to specify a base 8 (octal) number regardless of the current setting of *base*. Must precede any digits.

Additional notes

All numbers are stored as 24-bit signed integers. This allows a range of +8,388,607 to -8,388,608.

Cursor control keys work as they do in BASIC.

An Algorithm for 6510 Mnemonics

The challenge: to find the ideal mnemonic-to-opcode algorithm

by Glen C. Bodie

In the March 1987 issue of *Transactor*, Chris Miller wrote a very interesting article entitled "Assembling Assemblers". In his discussion of command look-up tables, he challenged "anyone to come up with an algorithm that will generate a unique, one byte value for every standard 6510 mnemonic." I love a good challenge! At first I thought it trivial, but it soon proved more complex than it first appeared. In general terms, the problem is to derive an algorithm that maps 56 unique three-character alphabetic strings into a unique value between 0 and 255 inclusive.

'...why not use the computer for what it is best at - dumb, repetitive searching.'

There are all sorts of simple algorithms that quickly come to mind. As an example, why not just add up the ASCII equivalents of each letter in the mnemonic? For all the examples, I'll use the mnemonic "LDA".

```
code = asc("l") + asc("d") + asc("a")
```

That is simple enough, but the CODE is not unique since "ADC" and "BCC" would produce the same result. Also the value of CODE ranges between 195 and 270 so it still needs some scaling. The easiest way to get CODE within range is to make each letter a value relative to "A". Now our algorithm looks like:

```
code = (asc("l") - asc("a")) + (asc("d") - asc("a")) +
    (asc("a") - asc("a"))
```

Now the range is 0 to 75. That's great, but the CODE is still not unique. So how are we going to make it unique? That's where it starts to get weird!

Base 10 and Base 26

Think about normal decimal numbers. In the value 444, the 4 has a different value in each position because it is multiplied by the base to some power, that is, by 1, 10 or 100. We can do the same thing with letters by multiplying each position by 1, 26 or 676. If we do that, the results are guaranteed to be unique, but the range is now 0 to 17576 and we need to map that into 0 to 255.

Before we get carried away with trying to find some mapping, think about how this will eventually be coded in the assembler. First of all, it will be written in machine language (ML). Multiplying by 26 in ML is a little awkward and time consuming. Instead of using 26 as the base, why not use 32 as the base since multiplying by 32 is the same as shifting left 5 times. The results will still be unique, but now the range is 0 to 32768!

We can try to map 32768 into 255 by simple techniques (such as ANDing with 255) or more complex techniques (such as XORing the high and low bytes), but these all turn our unique CODE into something that is no longer unique. It seems that the straightforward approaches just aren't going to work. A mathematical, analytical approach is beyond me, so why not use the computer for what it is best at - dumb, repetitive searching. What we want it to do is try a lot of alternative combinations of multipliers and transformations until it finds one that generates unique codes in the range 0 to 255.

What Computers do Best

To limit the problem, let's first make two simplifications:

1) Only use power of 2 multipliers
2) Use a transformation of the letters to reduce the combinations

For the transformation, there are only 14 possible first letters, 18 possible second letters and 15 possible third letters. All the combinations of these result in a CODE between 0 and 3780. The first step is to get the ASCII value of each letter, make it

'The program ran for over 30 hours before finding this answer.'

relative to "A", then use that in a table look-up to get a value between 0 and 14, 0 and 18 or 0 and 15. One key benefit of this is that we will discover very quickly if the mnemonic is invalid. Now our algorithm looks like this:

```
code = ( (table1("l"-"a")  * k1 +
          table2("d"-"a") * k2 +
          table3("a"-"a") * k3)) and 255
```

Without juggling the tables around and using K1=K2=K3=1, the code was not unique, so a little bit of searching was required. Program 1 does this. There are probably other answers and maybe even better ones, but this one works! The program checks through 13 different sets of multipliers and shifts all three tables through all the combinations. This comes out to a total of 49140 situations to check, calculating the CODE for each of the 56 mnemonics until it finds a duplicate or suceeds. The program ran for over 30 hours before finding this answer.

Program 2 is the resulting algorithm coded in BASIC and program 3 is the same thing in ML. The resulting CODE is a value between 0 and 255 which can be used in a table look-up to route the assembler to the correct processing, where the operand field could be parsed, the addressing mode determined, etc. Invalid mnemonics are found whenever the table results in a transformed value of zero. It is possible to input an invalid mnemonic that can be transformed without error into the same CODE as some valid mnemonic. This will have to be caught in the processing routine for each mnemonic by checking when you get there if the mnemonic really was what you were expecting.

There are two obvious alternatives to this solution:

1) a binary search tree
2) a hashing algorithm

The binary search is a classic programming technique for searching through ordered lists in the minimum number of comparisons. And though this program is a hashing algorithm of sorts, a true hashing algorithm would deal with duplicate codes, in case several mnemonics "hashed" to the same CODE. I don't know if Mr. Miller ended up using either of these. Both would work for sure, might be faster than this algorithm and have one large advantage - what if another mnemonic needs to be added to the list!

So, there it is - an algorithm to generate a unique code for each 6510 mnemonic. I sure love a good challenge.

Program 1: *This is the program that was used to generate the tables and constants so that the formula to find CODE (see text) would generate a unique value for each of the 56 opcode mnemonics. It takes 30 hours to find the solution.*

```
PG   100 rem----- program 1 ----------
MI   110 rem
FL   120 rem -- find the right algorithm --
AK   130 rem
PL   140 n=56: dim op$(n),op(n)
KF   150 for i=1 to n: read op$(i): op(i)=0: next
EK   160 data adc,and,asl,bcc,bcs,beq,bit,bmi
DJ   170 data bne,bpl,brk,bvc,bvs,clc,cld,cli
CH   180 data clv,cmp,cpx,cpy,dec,dex,dey,eor
FA   190 data inc,inx,iny,jmp,jsr,lda,ldx,ldy
EK   200 data lsr,nop,ora,pha,php,pla,plp,rol
BL   210 data ror,rti,rts,sbc,sec,sed,sei,sta
LM   220 data stx,sty,tax,tay,tsx,txa,txs,tya
PN   230 def fnh(x)=int(x/256)
EF   240 def fnl(x)=x-fnh(x)*256
JK   250 dim tr(2,25)
BD   260 for i=0 to 2: for j=0 to 25
PM   270 read tr(i,j): next j,i
IO   280 data  1,  2, 3,  4,  5,  0,  0,  0, 6,  7
AE   290 data  0,  8, 0,  9, 10, 11,  0, 12, 13, 14
EN   300 data  0,  0, 0,  0,  0,  0,  1,  2,  3,  4
KH   310 data  5,  0, 0,  6,  7,  0,  0,  8,  9, 10
IO   320 data 11, 12, 0, 13, 14, 15,  0, 16,  0, 17
BL   330 data 18,  0, 1,  0,  2,  3,  4,  0,  0,  0
GJ   340 data  5,  0, 6,  7,  0,  0,  0,  8,  9, 10
JM   350 data 11, 12, 0, 13,  0, 14, 15,  0
PP   360 dim k(12,2)
MF   370 for i=0 to 12: for j=0 to 2: read k(i,j): next j,i
BF   380 data 1,1,1,  1,2,4,   1,4,2
PG   390 data 2,1,4,  2,4,1,   4,2,1
PH   400 data 4,1,2,  1,4,16, 1,16,4
JF   410 data 4,1,16, 4,16,1, 16,4,1, 16,1,4
HK   420 for kk=0 to 12
BE   430 k1=k(kk,0): k2=k(kk,1): k3=k(kk,2)
CD   440 for a0=0 to 13
DE   450 for b0=0 to 17
NE   460 for c0=0 to 14
FJ   470 print "k1/k2/k3/a0/b0/c0="k1;k2;k3;a0;b0;c0
BD   480 gosub 2000: rem execute algo
DA   490 next c0, b0, a0, kk
AA   500 print "nothing worked!"
OP   510 end
HH   2000 rem ------------------------------
BF   2010 rem execute the algorithm
CA   2020 rem
CG   2030 xx=asc("a")
AA   2040 for i=1 to n
AD   2050 a$=left$(op$(i),1) : a=asc(a$)-xx
EF   2060 b$=mid$(op$(i),2,1): b=asc(b$)-xx
NE   2070 c$=right$(op$(i),1): c=asc(c$)-xx
AL   2080 a1=tr(0,a)+a0: a2=a1-int(a1/14)*14
          : if a2=0 then a2=14
FA   2090 b1=tr(1,b)+b0: b2=b1-int(b1/18)*18
```

```
      : if b2=0 then b2=18
GA    2100 c1=tr(2,c)+c0: c2=c1-int(c1/15)*15
      : if c2=0 then c2=15
LJ    2110 x = a2*k1 + b2*k2 + c2*k3
DA    2120 op(i) = fnl(x)
MB    2130 if i=1 then goto 2170
FD    2140 for j=1 to i-1
DF    2150 if op(i)=op(j) then gosub
CF    2160 next j
JF    2170 next i
NI    2180 print"it works!": end
```

Program 2: *This is a BASIC implementation of the algorithm calculated by Program 1. It prints each mnemonic along with the code generated; each code is unique, and could be used to direct an assembler to the appropriate parsing routine for the opcode.*

```
KA    100 rem --------- program 2 -----------
MI    110 rem
LK    120 n=56: dim op$(n),op(n)
GL    130 for i=1 to n: read op$(i): next
AJ    140 data adc,and,asl,bcc,bcs,beq,bit,bmi
PH    150 data bne,bpl,brk,bvc,bvs,clc,cld,cli
OF    160 data clv,cmp,cpx,cpy,dec,dex,dey,eor
BP    170 data inc,inx,iny,jmp,jsr,lda,ldx,ldy
AJ    180 data lsr,nop,ora,pha,php,pla,plp,rol
NJ    190 data ror,rti,rts,sbc,sec,sed,sei,sta
HL    200 data stx,sty,tax,tay,tsx,txa,txs,tya
AP    210 rem
GH    220 m=25: dim t1(m),t2(m),t3(m)
FI    230 for i=0 to m: read t1(i): next
IO    240 data  6,  7,  8,  9, 10,  0,  0,  0, 11
BE    250 data 12,  0, 13,  0, 14,  1,  2,  0,  3
DO    260 data  4,  5,  0,  0,  0,  0,  0,  0
AL    270 for i=0 to m: read t2(i): next
HF    280 data 10, 11, 12, 13, 14,  0,  0, 15, 16
DB    290 data  0,  0, 17, 18,  1,  2,  3,  0,  4
JD    300 data  5,  6,  0,  7,  0,  8,  9,  0
LN    310 for i=0 to m: read t3(i): next
EE    320 data  4,  0,  5,  6,  7,  0,  0,  0,  8
DC    330 data  0,  9, 10,  0,  0,  0, 11, 12, 13
OE    340 data 14, 15,  0,  1,  0,  2,  3,  0
MH    350 rem
EF    360 rem --- execute the algorithm
AK    370 rem  given an opcode mnemonic in 'op$',
IK    380 rem  this routine puts a corresponding
EK    390 rem  code in 'x'. the code is guaranteed
DA    400 rem  to be unique for each 6510 mnemonic.
DE    410 for i=1 to n: op$=op$(i)
EB    420 a=asc(left$(op$,1))-asc("a")
OH    430 a=t1(a): if a=0 goto 1000
CK    440 b=asc(mid$(op$,2,1))-asc("a")
LJ    450 b=t2(b): if b=0 goto 1000
CE    460 c=asc(right$(op$,1))-asc("a")
IL    470 c=t3(c): if c=0 goto 1000
JE    480 x=(a*1 + b*16 + c*4) and 255
HG    490 op(i) = x: next
CB    500 rem
AI    510 rem --- print results from array
GC    520 rem
DL    530 print" 6510 Mnemonic Algorithm"
OE    540 print
CM    550 for i=1 to 54 step 3
OB    560 for j=i to i+2
IH    570 print op$(j)"="right$("  "+str$(op(j)),3)"  ";
IJ    580 next: print: next
KM    590 print op$(55)"="right$("  "+str$(op(55)),3)"  ";
GE    600 print op$(56)"="right$("  "+str$(op(56)),3)
CG    610 end
EM    1000 rem --- invalid mnemonic
```

Program 3: *The machine-language implementation of the opcode algorithm in Program 2.*

```
NA    100 rem --------- program 3 -----------
MI    110 rem
GL    120 rem pal64 format source
AK    130 rem
IC    140 open 2,8,1,"0:output"
HG    150 sys 700
BL    160 .opt o2
LJ    170 *=  $c000
CI    180       jmp   begin
ED    190 ;
KK    200 ; table of mnemonics
IE    210 ;
JC    220 mnem      =*
DP    230       .asc "adcandaslbccbcsbeqbitbmibnebplbrk"
JL    240       .asc "bvcbvsclccldcliclvcmpcpxcpydecdex"
DP    250       .asc "deyeorincinxinyjmpjsrldaldxldylsr"
MP    260       .asc "noporaphaphpplaplprolrorrtirtssbc"
PC    270       .asc "secsedseistastxstytaxtaytsxtxatxstya"
OI    280 ;
LD    290 ; resulting op-codes
CK    300 ;
NG    310 ops  *=     *+56
GL    320 ;
ON    330 ; tables for transformations
KM    340 ;
OK    350 table1 =*
OL    360       .byte  6,  7,   8, 9, 10,  0, 0, 0, 11
HB    370       .byte 12, 0, 13, 0, 14,  1, 2, 0,  3
JL    380       .byte  4, 5,   0, 0,   0,  0, 0, 0, 0
GI    390
DO    400 table2 =*
HD    410       .byte 10, 11, 12, 13, 14,  0,  0, 15, 16
DP    420       .byte  0,  0, 17, 18,  1,  2,  3,  0,  4
JB    430       .byte  5,  6,  0,  7,  0,  8,  9,  0
IL    440
IB    450 table3 =*
OC    460       .byte  4,  0,  5,  6,  7,  0,  0,  0,  8
NA    470       .byte  0,  9, 10,  0,  0,  0, 11, 12, 13
ID    480       .byte 14, 15,  0,  1,  0,  2,  3,  0
AG    490 ;
LA    500 ; execute the algorithm
```

```
EH  510 ;
PH  520 begin     ldy #0
AG  530           sty opsx
GJ  540 loop1     lda mnem+0,y
HK  550           sec
AP  560           sbc asca
FA  570           tax
DA  580           lda table1,x
EI  590           beq error
HC  600           sta temp       ; (left-)*1
KK  610           lda mnem+1,y
NO  620           sec
GD  630           sbc asca
LE  640           tax
LE  650           lda table2,x
KM  660           beq error
HH  670           lsr
BI  680           lsr
LI  690           lsr
FJ  700           lsr
MD  710           clc
PC  720           adc temp
GO  730           sta temp       ; +(mid-)*16
OC  740           lda mnem+2,y
PG  750           sec
IL  760           sbc asca
NM  770           tax
PM  780           lda table3,x
ME  790           beq error
JP  800           lsr
DA  810           lsr
KK  820           clc
ML  830           adc temp       ; +(right-)*4
EF  840           ldx opsx
IO  850           sta ops,x
ID  860           inx
AL  870           stx opsx
AF  880           iny
KF  890           iny
EG  900           iny
PG  910           cpy #168
CG  920           bne loop1
FD  930           beq cont1
CC  940 ;
MH  950 ; error routine
GD  960 ;
KK  970 error     =*
KE  980 ;
DG  990 ; print results/process code byte
OF 1000 ;
NC 1010 cont1     =*
CH 1020 ;
EP 1030 ; data areas
GI 1040 ;
AE 1050 opsx      .byte 0
AI 1060 asca      .asc "a"
NB 1070 temp      .byte 0
EB 1080 .end
```

An Accurate TI$

Long-term timing accuracy for the C64 and C128

by Noel Nyman

TI$, and its companion numeric variable TI, are handy for timing applications. TI$ is easily reset and it's always there in BASIC. TI can also be read or set through the kernal RDTIM and SETTIM routines in machine language programs.

TI is incremented from the IRQ, or hardware interrupt routine. IRQ occurs about sixty times each second. IRQ adds 1 to TI each time it occurs. So, TI counts 1/60ths of a second, often called 'jiffies'.

TI is only as accurate as the IRQ frequency. Routines that halt interrupts also stop the TI/TI$ clock. Disk and tape accesses are the common causes of an inaccurate jiffy clock. But any machine language that contains an SEI (SEt Interrupt mask), will have the same effect.

There are two very accurate clocks in the C64 and C128, one in each CIA (Complex Interface Adaptor) chip. The clocks' timing is maintained by the power line frequency, and is accurate to a few millionths of a second each day. As the power load increases during the day, the frequency drops slightly. Your friendly power company monitors the frequency changes closely. Then, during non-peak hours, it increases the frequency a bit to compensate for the earlier changes.

The C64/C128 operating systems do not make use of either of the CIA clocks.

One of our clients wanted to use a C128 for industrial process control. The C64 and C128 are excellent cost effective choices for this type of application. They have an "open architecture", which means you can get at all the internal signals easily. Most industrial events occur in "real time", very slowly even to the ponderous 1MHz clock in the C64, so it's easy to monitor or control many machines at the same time with a very inexpensive, easily programmed computer. For more information on control and monitoring applications, see *Practical Interfacing Projects With The Commodore Computers*, by Robert Luetzow (TAB Books).

TI$ was not accurate enough for our client. The TOD (Time Of Day) clocks in the CIA chips seemed like a good choice to him. But reading and using them is much more difficult than using TI or TI$.

The TOD keeps time accurate to one-tenth of a second in four addresses or registers (see figure 1). The information is held in BCD (Binary Coded Decimal), a compact way of storing decimal digits. BCD complicates the task of using the time numerically.

Bit	7	6	5	4	3	2	1	0
56331 Hours	AM PM	x	x	H T	H U	H U	H U	H U
56330 Minutes	M T	M T	M T	M T	M U	M U	M U	M U
56329 Seconds	S T	S T	S T	S T	S U	S U	S U	S U
56328 Tenths	x	x	x	x	S N	S N	S N	S N
AM/PM Flag: AM=0, PM=1 T=tens digit U=units digit x=unused								

Bit Functions in the TOD Clock

Our client was monitoring the 'on time' of various plant machinery, and counting produced parts. His software combined time and counts in several statistical ways. He'd intended to use the TI counter, simply holding in memory the starting TI and the ending TI. Subtracting the two gives the total machine "on time" in jiffies.

To do that using the TOD requires extracting seven BCD digits and the AM/PM flag. Then the digits must be converted into tenths of seconds and added together. A different conversion is required to display the digits as time on the screen. What was needed was a way for the TOD clock to keep TI and TI$ accurate.

Maintaining An Accurate TI

Our approach was to modify the IRQ routine to monitor the TOD clock in CIA #1.

Since our client needed accuracy over a long period, we update TI/TI$ only once every hour. The new routine looks for minutes, seconds, and tenth-seconds in the TOD to all equal zero. When that occurs, the routine reads the TOD and updates the three bytes in zero page memory that make up TI. BASIC generates TI$ from TI whenever requested, so an accurate TI means an accurate TI$.

The "New-IRQ" routine gets the data for the TI update from three 25-byte tables. A value in each table corresponds to the value found in one of the three bytes of TI at the start of each hour. The routine determines the hour from the TOD hours byte and uses that value as an index to each of the three tables.

If you needed more accuracy, you could update TI each minute. That would require three tables of 1500 bytes each, much too large. You could use an algorithm to calculate the three values for TI. But in practice, an update every hour is accurate enough for real time applications.

Quirks in the TOD

There are only twenty-four hours in a day. A 25 byte table is needed because of an anomaly in the TOD clock. The TOD uses a twelve hour format, with an AM/PM flag, see figure #1. For example, 11 a.m. is represented by the value 17 ($11) stored in the hours register. Bit #4 is set for the 10's digit, for a value of 16. Bit #1 is set for the 1's digit.

For 11 p.m., the value is 145 ($91). Bits #4 and #0 are set. Bit #7 is set for PM, value 128. That's all very straightforward. The problem arises with 12 o'clock.

In twenty-four hour format, the time just after midnight is called '00' hours. In twelve hour format, we call it 12 a.m., so 12 noon (1200 hours) becomes 12 p.m. to us. The TOD, however, will clear the AM/PM bit if you try to set it with a 12. If you poke the hours register with a 146 to set 12 p.m., the TOD will change it to 18, its normal representation for 12 a.m.! If you POKE an 18, the TOD happily changes it to 146, or 12 a.m.

To set the TOD properly, we have to poke '00' for 12 a.m., and '18' for 12 p.m.. But when we read the TOD, an '18' in the hours register means twelve a.m., and a "146" means twelve p.m.

So, there are two possible values for 12 a.m., '0' if we've just set the TOD, and '18' if it's counted to a new day on its own. These added to the other twenty-three hour values make up tables of 25 bytes.

Using the New IRQ Routine

First, run the BASIC program "Create". A program named "New-IRQ" will be created on device #8. You can change the name or device number in line #1050. When prompted, enter the starting address in decimal. If there are errors in the DATA statements, you'll get an error message. Scratch "New-IRQ", correct the DATA errors, and run "Create" again.

"Create" is a complex generator because the "New-IRQ" routine is not relocatable. There are several addresses and vectors that will change, depending on where you decide to place the code.

You need this flexibility. "New-IRQ" will work on either a C64 or a C128. But the 'safe' places to store machine language code vary between the two machines. "New-IRQ" uses 186 bytes. Anywhere in the area between 49152 and 53247 ($C000-$CFFF) is good on the C64. On a C128, the area starting at 4864 ($1300) may be safe. The tape buffer and RS232 buffers are good choices, if your software doesn't use them. In any case, keep the code below 16384 ($4000), so the new routine will be 'visible' regardless of what bank you're in when an IRQ occurs.

Line #1110 compensates for a bug in the 65xx/85xx family of processors. Once "New-IRQ" is finished, the standard IRQ routine is executed by an indirect jump, JMP ($C0FF) in machine code. The actual address jumped through will depend on where you located the "New-IRQ". The indirect vector may span a page boundary. In our example above, the vector sits at $C0FF and $C100. When this happens, the indirect jump will not work properly (see *6502 Assembly Language Programming*, by Lance Leventhal, pages 3-13).

Line #1110 checks for this possibility and, if found, adds a NOP code to the start of "New-IRQ". This makes the code one byte longer, but avoids the bug under all circumstances.

LOAD "New-IRQ",8,1 either before you run your main program or from within the program itself. Use SYS xxxxx (where xxxxx is the decimal starting address you used with "Create") to initialize the routine.

The program "BASIC Time" sets TI$ and the TOD to the current time, taking care of the 12 o'clock anomaly. Add "BASIC Time" as a subroutine to your program, and call it early.

You can also use "New-IRQ" as a stand-alone program to keep TI$ accurate in direct mode. Be sure to use "BASIC Time" to set the clocks, changing the RETURN in line #50150 to END.

In either program or direct mode, pressing the RESTORE key will disable "New-IRQ". A SYS to the starting address will re-enable it. Although you can call "BASIC Time" again, it's probably not necessary. Unless the computer is shut off, the TOD and TI will continue running after RESTORE is pressed.

Listing 1: "BASIC Time" subroutine

```
MP  50000 rem subroutine to get time from
BL  50005 rem user and convert for ti$ and
LE  50010 rem the tod clock in cia #1.
DN  50015 :
AN  50020 rem by noel nyman
NN  50025 :
CH  50030 rem called as a subroutine
HO  50035 :
JE  50040 rem uses the following variables
EC  50045 rem  ap - am/pm flag, am=0 pm=1
GM  50050 rem  ht - hours, used for tod
EH  50055 rem  mt - minutes, used for tod
GA  50060 rem  sc - seconds, used for tod
KE  50065 rem  tt - temporary variable
GG  50070 rem  tt%- temporary variable
EL  50075 rem  ht$- hours for ti$
MO  50080 rem  mt$- minutes for ti$
IN  50085 rem  st$- seconds for ti$
DA  50090 rem  tt$- temporary for ti$
FD  50095 rem  ap$- temporary variable
HC  50099 :
CN  50100 print "current time is " left$(ti$,2) ":" mid$(ti$,3,2) ":";
BH  50110 print right$(ti$,2)
FD  50120 print: print "enter new time, or [rvs]  [rvs off] to quit"
ID  50130 print: print "enter new hours (0-23): ";
GP  50140 open 9,0: input#9,ht$: close9
MJ  50150 if ht$="" then return
MI  50160 ht=val(ht$): if ht<0 or ht>23 goto 50130
MI  50170 ap=0-(ht>12)
IH  50180 :
EC  50200 print: print "enter new minutes (0-59): ";
GE  50210 open 9,0: input#9,mt$: close9
IP  50220 mt=val(mt$): if mt<0 or mt>59 goto 50200
DL  50230 if len(mt$)<2 then mt$="00"+mt$
EL  50240 :
HE  50300 print: print "enter new seconds (0-59): ";
GL  50310 open 9,0: input#9,st$: close9
GA  50320 sc=val(st$): if sc<0 or sc>59 goto 50300
LD  50330 if len(st$)<2 then st$="00"+st$
IB  50340 :
AO  50400 if ap goto 50500
GJ  50410 print: if ht<12 goto 50460
BI  50420 print "am or noon (a/n)?";
DH  50430 get ap$: if ap$="" goto 50430
IL  50440 if ap$<>"a" and ap$<>"A" and ap$<>"n" and ap$<>"N"
    goto 50430
FE  50450 ap=0-(ap$="n")-(ap$="N"): goto 50500
HB  50460 print "am or pm (a/p)?";
DK  50470 get ap$: if ap$="" goto 50470
GP  50480 if ap$<>"a" and ap$<>"A" and ap$<>"p" and ap$<>"P"
    goto 50470
IK  50490 ap=0-(ap$="p")-(ap$="P")
JB  50500 ht=ht+12*((ht=12) and (ap=0))
NC  50510 ht=ht-12*((ht<12) and (ap=1))
AM  50520 ht$=str$(ht): ht$="00"+right$(ht$,len(ht$)-1)
IF  50530 tt$=right$(ht$,2)+right$(mt$,2)+right$(st$,2): ti$=tt$
KH  50540 tt=0: if ht>12 then ht=ht-12: tt=128
LJ  50550 tt%=ht/10: tt=tt+(16*tt%)+(ht-10*tt%)
FP  50560 poke 56335,peek(56335) and 127
HA  50570 poke 56331,tt
PD  50580 tt%=mt/10: tt=(16*tt%)+(mt-10*tt%)
IB  50590 poke 56330,tt
AB  50600 tt%=sc/10: tt=(16*tt%)+(sc-10*tt%)
FE  50610 poke 56329,tt
AI  50620 poke 56328,0
CO  50630 print: goto 50100
```

Listing 2: "Create"

```
CN  1000 rem ** this program will create a
OJ  1010 rem ** machine language program
IH  1020 rem ** which modifies the irq
JK  1030 rem ** routine to set  ti$=tod cia#1
EI  1040 :
HG  1050 open15,8,15: open8,8,1,"0:new-irq": ck=0
JK  1060 input#15,e,e$,b,c: if e then close15: print e;e$;b;c:stop
GB  1070 input "starting address: ";s$
KC  1080 s=val(s$): if s<1 goto1090
EO  1090 def fnh(x)=int(x/256): def fnl(x)=x-256*int(x/256)
IO  1100 print#8,chr$(fnl(s));:print#8,chr$(fnh(s));
PH  1110 if fnl(s+25)=255 then print#8,chr$(234);:s=s+1
GA  1120 for x=1 to 5: read a: ck=ck+a: print#8,chr$(a);: next
EB  1130 print#8,chr$(fnl(s+25));:print#8,chr$(fnh(s+25));
HB  1140 for x=1 to 4: read a: ck=ck+a: print#8,chr$(a);: next
KC  1150 print#8,chr$(fnl(s+26));:print#8,chr$(fnh(s+26));
HE  1160 print#8,chr$(169);chr$(fnl(s+27));
    chr$(141);chr$(20);chr$(3);
DC  1170 print#8,chr$(169);chr$(fnh(s+27));
AJ  1180 for x=1 to 66:read a: ck=ck+a: print#8,chr$(a);:next
KG  1190 print#8,chr$(fnl(s+112));:print#8,chr$(fnh(s+112));
AF  1200 for x=1 to 3: read a: ck=ck+a: print#8,chr$(a);:next
NJ  1210 print#8,chr$(fnl(s+137));:print#8,chr$(fnh(s+137));
EG  1220 for x=1 to 3: read a: ck=ck+a: print#8,chr$(a);:next
BK  1230 print#8,chr$(fnl(s+162));:print#8,chr$(fnh(s+162));
NK  1240 for x=1 to 12: read a: ck=ck+a: print#8,chr$(a);:next
MI  1250 print#8,chr$(fnl(s+25));:print#8,chr$(fnh(s+25));
PN  1260 for x=1 to 75: read a: ck=ck+a: print#8,chr$(a);:next
HG  1270 close8:close15
DJ  1280 if ck<>18314 then print"---error in data statements!---":
    end
JJ  1290 print "***irq module created***": end
II  1300 :
KB  1310 data 120, 173,  20,   3, 141, 173,  21,   3
JD  1320 data 141, 141,  21,   3,  88,  96,   0,   0
GO  1330 data 165, 251,  72, 165, 252,  72, 173,  11
DH  1340 data 220, 133, 251, 173,  10, 220, 208,  57
JC  1350 data 173,   9, 220, 208,  52, 173,   8, 220
DO  1360 data 208,  47, 165, 251,  41,  15, 133, 252
OG  1370 data 165, 251,  41,  16, 240,   7,  24, 169
HK  1380 data  10, 101, 252, 133, 252, 165, 251,  16
LH  1390 data   7,  24, 169,  12, 101, 252, 133, 252
MM  1400 data 164, 252, 185, 133, 160, 185, 133, 161
MJ  1410 data 185, 133, 162, 173,   8, 220, 104, 133
CJ  1420 data 252, 104, 133, 251, 108,   0,   3,   6
AJ  1430 data   9,  13,  16,  19,  23,  26,  29,  32
GO  1440 data  36,   0,  42,  46,  49,  52,  56,  59
KP  1450 data  62,  65,  69,  72,  75,  39,   0,  75
LI  1460 data 151, 227,  47, 122, 198,  18,  94, 169
JA  1470 data 245,  65,   0, 216,  36, 112, 188,   7
JP  1480 data  83, 159, 235,  54, 130, 206, 141,   0
JA  1490 data 192, 128,  64,   0, 192, 128,  64,   0
DB  1500 data 192, 128,  64,   0, 192, 128,  64,   0
NB  1510 data 192, 128,  64,   0, 192, 128,  64,   0
```

Listing 3: "New-IRQ"

```
*****************************
*       this routine is added to the *
*       normal irq to reset the      *
*       system clock (ti and ti$)    *
*       to the time-of-day clock     *
*       in cia #1.                   *
*                                    *
*       the ti/ti$ clock is subject  *
*       to accumulated errors,       *
*       especially during disk       *
*       and tape access. the tod     *
*       clock accuracy is            *
*       maintained by the power      *
*       line frequency.              *
*                                    *
*       this routine sets the        *
*       ti/ti$ clock = to the        *
*       tod clock on the tod         *
*       hour (minutes and            *
*       seconds all = zero).         *
*                                    *
*       noel nyman 8/87              *
*****************************

ti        = $a0       ;first byte of ti
temp1     = $fb       ;temporary storage, original value unchanged
temp2     = $fc       ;temporary storage, original value unchanged

irqvec    = $0314     ;address of irq vector

hours     = $dc0b     ;cia #1 hours register
minutes   = $dc0a     ;cia #1 minutes register
seconds   = $dc09     ;cia #1 seconds register
tenths    = $dc08     ;cia #1 tenths sec register

* routine can be placed at any convenient location.

          org $c000

* get the current irq vector and store it in
* 'holdirq.' place the vector to the added
* code at the irq vector address.
start     sei
          lda  irqvec
          sta  holdirq
          lda  irqvec+1
          sta  holdirq+1
          lda  #<newirq
          sta  irqvec
          lda  #>newirq
          sta  irqvec+1
          cli
          rts
holdirq   hex 00,00
          ;code added to irq routine starts here
newirq    lda  temp1    ;store current values of
          pha           ;temp1 and temp2 on stack
          lda  temp2    ;so we can restore them in case
          pha           ;the interrupted application uses them
          lda  hours    ;reading the hours register
          sta  temp1    ;halts the clock, we store
*                       ;the value in temp1, just
*                       ;in case it's time to use it
          lda  minutes  ;check for minutes = zero
          bne  exit     ;if not zero, not time to update
          lda  seconds  ;check for zero seconds
          bne  exit     ;and skip update if not zero
          lda  tenths   ;check for zero tenths of a second
          bne  exit     ;and skip update of not zero
*              ;the update routine converts the
*              ;value in the hours register
*              ;from bcd/am-pm format into
*              ;a binary number in the range
*              ;0-25, where 0 or 12 equals 12am,
*              ;1=1am, 13=1pm, and 25=noon.
*              ;the converted number is used as an
*              ;index to a table to store the
*              ;proper values in the three bytes of ti
          lda  temp1    ;hours register value
          and  #$0f     ;mask out upper bcd digit, am-pm
          sta  temp2    ;store lower hours digit
          lda  temp1    ;get hours value again
          and  #$10     ;mask out all but the upper bcd digit.
*                       ;this can only be zero or one.
          beq  skipten  ;if zero, don't add 10 to temp2
          clc
          lda  #$0a     ;if high hours digit was one,
          adc  temp2    ;add 10 to temp2
          sta  temp2
skipten   lda  temp1    ;get hours value again
          bpl  skipap   ;if high bit clear
*                       ;time is am, skip the routine
*                       ;that adds 12 for pm times
          clc
          lda  #$0c     ;add 12 to temp2
          adc  temp2    ;if time is pm
          sta  temp2
skipap    ldy  temp2    ;put index to table in y
          lda  table1,y ;get values from
          sta  ti       ;three tables
          lda  table2,y ;and store in the
          sta  ti+1     ;three bytes
          lda  table3,y ;of ti
          sta ti+2
*              ;restart display,
*              ;restore the previous values in
*              ;temp1 and temp2, and jump
*              ;through the stored irq vector to
*              ;complete the irq routine.
exit      lda  tenths   ;read tenths to restart display
          pla
          sta  temp2
          pla
          sta  temp1
          jmp (holdirq)

*         the following three tables hold the values normally
*         found in the three bytes of ti at the 'top' of each hour.
table1    hex 00,03,06,09,0d,10,13,17,1a,1d,20,24
          hex 00,2a,2e,31,34,38,3b,3e,41,45,48,4b,27
table2    hex 00,4b,97,e3,2f,7a,c6,12,5e,a9,f5,41
          hex 00,d8,24,70,bc,07,53,9f,eb,36,82,ce,8d
table3    hex 00,c0,80,40,00,c0,80,40,00,c0,80,40
          hex 00,c0,80,40,00,c0,80,40,00,c0,80,40,00
```

Olsen's Raid

An Update to the "Shiloh's Raid" Relative File Bug Fix

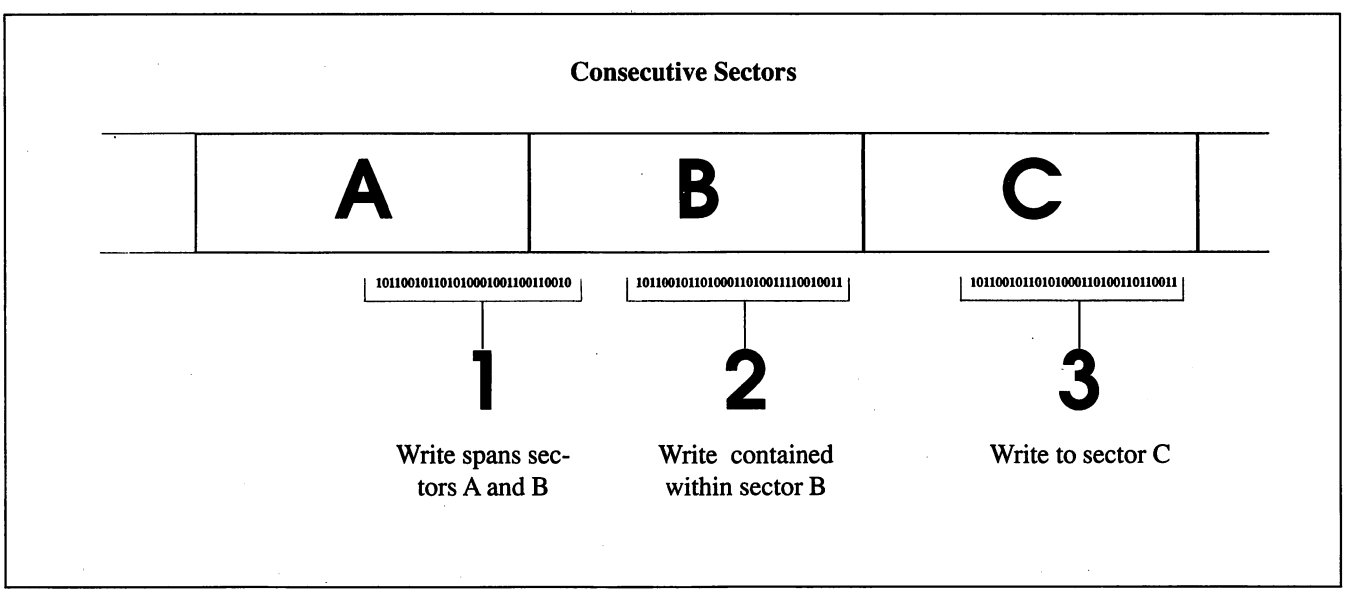
Volume 7, Issue 4 of *Transactor* contained an article by David Shiloh called "Shiloh's Raid" that claimed to eliminate the dreaded relative file bug: it showed under what circumstances the bug occurred, and took extra precautions when writing to the file under those conditions. Since that article appeared, we heard from Helen Olsen, who found a flaw in Shiloh's explanation of when the bug occurs. Helen sent us several programs to illustrate her point, and after hearing from her, and again from David Shiloh, we think it's time to clarify things a bit.

First, let's back-track a little. A generally-known bug in the 1541 causes problems under rare conditions when writing to a relative file. The fix (also generally known) is easy - just position the record pointer *twice* before writing a relative file record. The extra point (and some say a short delay as well) seems to eliminate the bug, so most experts advise to handle relative files in this way and eliminate the problem. David Shiloh took the extra step of finding under exactly what conditions the bug occurs, and applying the fix only under those conditions. The program he presented along with the article was supposed to prove that the fix works by checking for errors in a long random-write test both with and without the "Shiloh's Raid" routine in place.

Shiloh explained that the bug occurs under the following conditions - refer to the diagram below to illuminate the explanation: data is written to a relative file record, "spilling over" from one sector to another. This is write number 1 in the diagram, spanning sectors A and B. Write number 2 then takes place, to a record residing wholly within the next contiguous sector - sector B. According to Shiloh, the bug is now waiting to happen, and if a write (3) now occurs to the next sector (C), the data is instead erroneously written to sector A, potentially spilling into sector B as well.

Shiloh's solution was to detect when this condition was about to occur and apply the standard fix, to position the record pointer twice and pause before writing. The advantage to Shiloh's Raid is that the double-point need only be done on the rare occasions when the above circumstances occur, saving time for typical relative file access.

Enter Helen Olsen. Her main point was that the conditions that Shiloh sets for the bug to occur are too strict; only writes 1 and 3 (referring to the diagram again) need to take place to trigger the bug. She suggested that positioning an extra time (without the delay) after writing to a split record (1) is the best solution.



As it turns out, David Shiloh's explanation of when the bug occurs *is* too exacting, but the 'Shiloh's Raid' program works properly, that is, it senses trouble and does the extra pointer positioning even when only writes '1' and '3' occur. David Shiloh told us this, and shortly thereafter, Helen Olsen sent us a letter that concurred. Here is part of that letter:

> "To sum up my position on Shiloh's Raid: His description of the *cause* of the bug is wrong - his 1, 2, 3 sequence, with 2 setting up the bug is wrong. The bug happens with 1 followed by 3. *I* was wrong about his fix not working because I assumed that it was applied only in response to the 1, 2, 3 sequence. He may not realize it, but his subroutine repeats the positioning command often outside that sequence, including the 1, 3 sequence, which may be why he's unaware that *it* causes the bug, also... His fix is also applied when it is not necessary... In random use of a relative file, my fix, which repeats the position command after every write to a split record, will surely be used unnecessarily more often than Shiloh's fix, but since I don't know the cause of the bug (nor does he), I feel safer using it."

Helen presented programs along with her letters that proved her point, showing that only writes 1 and 3 were required to cause the bug, and showing that her fix worked.

So where does that leave us at the Transactor? Well, with a certain amount of egg on our face, to begin with, for not verifying that Shiloh's program was doing exactly what the text of the article said it was supposed to do. It may be even worse than that (for us), because the version of Shiloh's Raid that was printed was much improved cosmetically from the original - expanding from nineteen lines of tightly packed, unreadable code, to a page and a half of commented BASIC that had a chance of being understood. Perhaps something was lost in the translation that accounted for the application of the extra point-and-wait when it was unnecessary, i.e. outside of the "1, 2, 3" and "1, 3" conditions. In any case, we perhaps got carried away a tad in presenting the last word on the relative file bug, and we're glad that Helen brought us down to reality, though "glad" may not be the most appropriate word all around.

So, to summarize: The original Shiloh's Raid program was correct, in that it prevented the bug from occurring, and only applied the fix on a very small percentage of writes to the file. Shiloh's explanation of when the bug occurs doesn't cover all situations, so is only partially correct. Helen Olsen is right about the flaw in Shiloh's explanation, and her solution also seems to stop the bug from occurring. There is still no proof that there are no other ways in which the bug can occur, but it seems that you are safe if you use Shiloh's Raid, Olsen's fix, or if you just position twice before every write to a relative file.

Thus closes the file on Shiloh's Raid; as Helen Olsen ended her letter after announcing that it was to be her last on the subject, "Did I hear a heartfelt 'amen'?"

Three Movers for the C64

Relocatable utilities to save and restore programs, screens and colour data

by Richard Curcio

Many video effects are possible on the C64 by quickly moving large blocks of memory. The routines presented here are designed to do just that. These routines are relocatable; simply change the variable SA in the associated Basic loader to the desired address. The FOR-NEXT loop value in each loader indicates the length of the ML. Although relocatable code is often longer than non-relocating, this can be justified by the ability to place it wherever there is enough room, without modification.

Move.Plus

The first routine is not limited to moving video or graphic information. It can move any chunk of memory to any location. This routine has the ability to retrieve data stored in RAM "under" the BASIC or Kernal ROMs. The syntax for calling the routine is:

```
sys MOVE, source address, destination address,
  number of bytes, mask
```

where MOVE is the start address of the routine (51200 in the version listed). The last value, mask, must be in the range 0 to 255. If mask is greater than zero, interrupts are disabled and BASIC and the Kernal ROMs are switched out, allowing the RAM beneath them to be accessed. The memory configuration in effect when the routine was called is restored upon return. Thus, if a modified BASIC in RAM is in effect, the system stays that way. The RAM under the Kernal would still be available for storage in that case.

To copy the BASIC ROM into underlying RAM:

```
sys MOVE, 40960, 40960, 8192, 0 : poke 1, 54
```

Note that mask value zero is used so that ROM may be read. Memory writes to addresses in ROM always "fall through" to the RAM beneath.

This routine can be used to fill memory by poking the first address with the desired value and moving up by one location.

```
poke 55296, 0:sys MOVE, 55296, 55297, 999, 0
```

...will change all characters on the screen to black by filling color memory with zeroes.

Character ROM is not affected by the mask value. To copy character ROM into RAM requires almost as much code as doing it in BASIC. It happens several hundred times faster, though:

```
100 poke 56334, peek(56334) and 254
    : rem disable interrupts
110 poke 1, peek(1) and 251
    : rem switch in character rom
120 sys MOVE, 53248, 12288, 2048, 0
    : rem move upper case/graphics character set to ram
130 poke 1, peek(1) or 4
    : rem switch out character rom
140 poke 56334, peek(56334) or 1
    : rem enable interrupts
```

Further pokes are necessary to protect the new character set from Basic variables and to tell the VIC chip where to find the characters. Consult the *Programmer's Reference Guide* or other sources for more information on C-64 graphics.

If the number of bytes to be moved causes the destination address to "roll over" from $FFFF to $0000, the remaining bytes are not moved and the routine returns to BASIC. The value 255 is left in location 782 (SYREG) to indicate that the move was incomplete. Caution should be used when specifying addresses below 828 as destinations.

Color.Move

This routine will save the contents of colour memory to one of sixteen sections "under" the Kernal ROM. Because colour RAM consists of 1024 nybbles, each color map is compacted into 512 bytes.

To save a color map:

```
sys COLOR, section #
```

...where COLOR is the address of the routine (51320 in the listed version) and section # is 0 to 15.

To recall a particular color map:

sys COLOR+4, section #

To store colors under the BASIC ROM, 'poke COLOR+19,160'. To change back to the Kernal, 'poke COLOR+19,224'. This makes a total of 32 colour maps available. If a section were to roll over to zero page (due to poking too large a value into COLOR+19), the routine stops with '?illegal quantity' before any transfer takes place.

It is also possible to reduce the number of storage sections. 'Poke COLOR+12,8' would limit storage to sections zero through seven.

Video.Move

This routine will save screen and color memory to one of five sections under the Kernal. All 1024 bytes of the screen are saved, including the sixteen unused bytes and the eight bytes of sprite data pointers. As in 'Color.Move', colour ram is compacted into 512 bytes for a total of 1536 bytes per section.

To save screen and color:

sys VIDEO, section, screen org

where VIDEO is the address where the routine is located (51456) and section is 0 to 4. The last parameter, screen org, determines where the screen, which is also called the video matrix, resides. If screen org is zero, the routine uses the contents of location 648 to determine where the screen is located. The operating system uses this location to determine where the screen is for text output. If screen org is greater than zero, the routine uses the contents of the VIC-II chip and data port A of CIA 2 to determine the video matrix location, which is the screen currently displayed. When the C-64 is in high-resolution mode, each byte of screen memory contains the background and foreground colors for the hi-res bit map. The value in location 648 in this case is irrelevant, and the value the VIC chip uses may be completely different. (Color ram is also irrelevant unless the C-64 is in multi-color hi-res.) In other words, an org value greater than zero uses the currently displayed video matrix. A value of zero uses the text screen, which may or may not be visible.

To recover screen and color:

sys VIDEO+4, section, screen org

BASIC and Kernal ROMs are switched out during both storage and retrieval.

Since 'Video.Move' stores information under the Kernal, section #4 should be used with caution due to the writing to RAM that takes place when RUN/STOP-RESTORE is pressed. To demonstrate, list something to screen in one color and save it to section 4. Press RUN/STOP-RESTORE to clear the screen and then recall section 4. The lower portion of the screen will have characters of different colors. This warning also applies when using section #14 under the Kernal with 'Color.Move'.

As with 'Color.Move', it is possible to change the storage area. 'Poke VIDEO+16,160' will change the storage area to RAM under the BASIC ROM. 'Poke VIDEO+16,224' to store under the Kernal. Like 'Color.Move', the routine stops with '?illegal quantity' if a section will roll over to zero page.

It is also possible to copy from any given screen location. Say you have a bit-map at $6000 with its color matrix at $5C00. If for some reason you want to store the bit-map colors while maintaining text mode, poke the high byte of the bit-map colour matrix into location 648.

```
200 TX = peek(648): rem current text screen
210 VM = 92: rem high byte of bit-map colors at $5C00
220 poke 648, VM: sys VIDEO, section #, 0
    : rem use loc. 648 as org
230 poke 648, TX: rem restore normal text
```

This is risky. If the program should stop with an error before location 648 is restored to its previous contents, the machine will appear to have crashed. In reality, the system is printing to the bit-map colour matrix but the VIC chip is still displaying the old text screen.

Simple Windows

If you need a simple-minded Window utility, 'Video.Move' combined with a PRINT "at" routine (that allows you to print to any given screen position) can do a good simulation. Just store the current text and colors, use PRINT@ or some other means to display a box with the desired message, then recall the screen contents to make the "window" disappear. Or store up to five screens for overlapping windows. If it is not necessary to store the colours, 'Move.Plus' can store eight screens under ROM.

One factor complicates this pseudo-window application. When printing to the screen, the system keeps track of which screen lines wrap around to the line below. This information is stored in 25 locations in zero page. When the screen is POKEd, which is what 'Video.Move' or 'Move.Plus' do, this link table is not updated. A recalled screen would have the links of the previous screen. This may or may not be a problem, depending on what you do with the recalled screen. PRINTs may not come out as expected. INPUT from the screen may be affected as well. To get around this problem, Move.Plus can be used to store the contents of the link table in some safe location.

sys MOVE, 217, destination, 25, 0

To recall the screen links, 217 becomes the destination and the proper mask value should be used if the links were stored under ROM. Your program would have to keep track of

which stored screen matches a stored link table.

Other Uses

'Video.Move' and 'Color.Move' can create interesting effects in low-res and even achieve pseudo animation by storing and retrieving a number of alternate screens and/or colors. 'Video.Move' will change the sprite data pointers along with the screen's contents. If enough RAM is available, 'Move.Plus' can store and retrieve a number of bit-maps. 'Video.Move' will store both color maps of hi-res multi-color graphics. 'Move.Plus' can also redefine custom characters or sprites for another form of animation.

One of the most useful applications for 'Move.Plus' is to store a number of machine language routines that need to be at the same address. Load a program to its designated location, then use 'Move.Plus' to put it someplace safe. Repeat for other programs, then retrieve each routine as needed. (Don't try this with "wedges", though.) 'Move.Plus' can even move itself and because of its relocatability it will function at its new location.

Listing 1: Assembler source code for 'Move.Plus'

```
JO 1000          *=  $c800
PH 1030 ;---------- move.plus ----------
GI 1040 ;
KA 1050 frmnum =  $ad8a
PA 1060 chkcom =  $aefd
FN 1070 getadr =  $b7f7
ED 1080 srce   =  $c3
JM 1090 nbytes =  $14
AF 1100 dest   =  $c1
MM 1110 ;
IM 1120 ; get and store source and destination
AO 1130 ;
GJ 1140 begin  jsr  chkcom
LE 1150        jsr  frmnum
EJ 1160        jsr  getadr
MG 1170        sty  srce
CC 1180        sta  srce+1
EC 1190        jsr  chkcom
NH 1200        jsr  frmnum
GM 1210        jsr  getadr
BO 1220        sty  dest
HJ 1230        sta  dest+1
GF 1240        jsr  chkcom
IF 1250 ;
MG 1260 ; get number of bytes and mask value
MG 1270 ;
AP 1280        jsr  $b7eb      ; two bytes in $14/15, one
byte in x
AI 1290 ;
FF 1300 ; $14-15 has number of bytes
EJ 1310 ;
MN 1320        txa
JH 1330        beq  calc       ; if mask = 0
FN 1340        sei
PM 1350        lda  $01        ; get mem. config.
MM 1360        pha
KB 1370        and  #$fd       ; mask basic and kernel
BC 1380        sta  $01
EO 1390 ;
FD 1400 ; calculate end address
IP 1410 ;
PN 1420 calc   clc             ; add #bytes to dest.
GB 1430        lda  nbytes
BB 1440        adc  dest
IG 1450        sta  nbytes
GP 1460        lda  nbytes+1
LD 1470        adc  dest+1
LK 1480        sta  nbytes+1   ; $14/15= dest.end+1
FF 1490        ldy  #$00
DJ 1500 start  lda  (srce),y
FO 1510        sta  (dest),y
MJ 1520 bump1  inc  srce
BM 1530        bne  bump2
MG 1540        inc  srce+1
GM 1550 bump2  inc  dest
CK 1560        bne  comp
NM 1570        inc  dest+1
JD 1580        beq  rollo      ; if dest. rolls over
IL 1590 comp   lda  dest+1
HM 1600        cmp  nbytes+1
GC 1610        bne  start
DN 1620        lda  dest
DC 1630        cmp  nbytes
EE 1640        bne  start
KD 1650 done   txa             ; was mask = 0
LD 1660        beq  exit       ; yeah
HH 1670        pla             ; restore mem. config.
NE 1680        sta  $01
IC 1690        cli
KJ 1700 exit   rts
EC 1710 ;
AE 1720 rollo  dey             ; leave 255 in
NC 1730        bne  done       ; location 782
```

Listing 2: 'Color.Move'

```
OB 1000          *=  $c878
ON 1030 ;--- color.move ---
GI 1040 ;
EC 1050 temp   =   $c3
KJ 1060 ;
KA 1070        ldy  #$ff       ; flag = store
HD 1080        bne  setup
CH 1090        ldy  #$00       ; flag = recover
JD 1100 setup  sty  temp       ; save entry
NP 1110        jsr  $b7f1      ; section # in .x
OC 1120        cpx  #$10       ; chk. range 0-15
GE 1130        bcs  qty1       ; >$0f illegal
IC 1140        txa
MJ 1150        asl             ; times 2
OP 1160        clc
GK 1170        adc  #$e0       ; use a0 for bas.rom
MG 1180        bcs  qty1       ; rolled over. no room
BH 1190        tax
MI 1200        inx
PP 1210        beq  qty1       ; will roll over. no room
DE 1220        ldx  #$00
PC 1230        sta  $25        ; init. addresses
IP 1240        stx  $22
IA 1250        stx  $24
AB 1260        ldx  #$fe       ; counter
NN 1270        ldy  temp       ; which way
OE 1280        beq  recover
ME 1290 savcol iny             ; #$ff+1
EJ 1300        lda  #$d8       ; hb of color ram
FN 1310        sta  $23        ; $22-23 = source
JN 1320 col1   lda  ($22),y    ; get a nybble
NN 1330        asl
```

```
HO 1340          asl
BP 1350          asl
CE 1360          asl             ; move to hi nybble
DJ 1370          sta  temp       ; save it
EN 1380          inc  $23        ; next pg. of col. mem.
LE 1390          lda  ($22),y    ; get it
LI 1400          and  #$0f
PK 1410          ora  temp       ; combine 'em
JD 1420          dec  $23        ; prepare for next
OJ 1430          sta  ($24),y
AI 1440          iny
FL 1450          bne  col1
AJ 1460          inx
JG 1470          beq  exit       ; enough times
HH 1480          inc  $25
LH 1490          inc  $23
FI 1500          inc  $23
OM 1510          bne  col1       ; branch always
GG 1520  ;
PM 1530  qty1    jmp  $b248      ; illegal quant.
KH 1540  ;
HF 1550  recover sei
GN 1560          lda  $01        ; get config.
OJ 1570          pha
MF 1580          and  #$fd       ; mask out roms
DP 1590          sta  $01
OA 1600          lda  #$d9       ; pg2 of color ram
BB 1610          sta  $23
KA 1620  col2    lda  ($24),y    ; get a byte
LM 1630          sta  ($22),y    ; ignore hi-nybble
BE 1640          lsr
LE 1650          lsr
FF 1660          lsr
MO 1670          lsr             ; move to lo nyb
EB 1680          dec  $23
MB 1690          sta  ($22),y    ; store it
NE 1700          inc  $23
DB 1710          iny             ; pointer
HM 1720          bne  col2
OJ 1730          inx
EA 1740          beq  rdone
FI 1750          inc  $25
JI 1760          inc  $23
DJ 1770          inc  $23
DA 1780          bne  col2
GL 1790  rdone   pla             ; get config
DK 1800          sta  $01        ; and restore it
AK 1810          cli
CB 1820  exit    rts
```

Listing 3: 'Video.Move'.

```
LO 1000  *       =    $c900
IK 1030  ;------ video.move ------
GI 1040  ;
EC 1050  temp    =    $c3
KJ 1060  ;
PB 1070          ldy  #$ff       ; here to store
HD 1080          bne  setup
HI 1090          ldy  #$00       ; here to recover
DK 1100  setup   sty  temp
PO 1110          jsr  $b7f1      ; one byte in x
NC 1120          cpx  #$05       ; section 0-4
LM 1130          bcs  qty1
NL 1140          lda  #$e0       ; use #$a0 for bas.rom
GP 1150  again   dex
PJ 1160          bmi  cont       ; no addition
IA 1170          clc
JM 1180          adc  #$06
OM 1190          bcs  qty1       ; no good
GD 1200          bne  again
PC 1210  cont    tay             ; save result
PK 1220          adc  #$05       ; enough room
MA 1230          bcc  ok
OE 1240  ;
BL 1250  qty1    jmp  $b248      ; illegal quantity
CG 1260  ;
BB 1270  ok      sty  $15        ; hi-byt of sect. #
EM 1280          lda  $0288      ; hi-byt of screen loc.
ML 1290          sta  $af        ; init. addresses
HD 1300          lda  #$00
JD 1310          sta  $ae
AP 1320          sta  $14
AD 1330          jsr  $b7f1      ; which screen org.
AP 1340          txa
KA 1350          beq  movit      ; 0 = text screen
PJ 1360          lda  $dd00      ; vid.bank from cia 2
FC 1370          ror             ; bits 0 & 1
IM 1380          ror             ; into 6 & 7
CC 1390          ror
BA 1400          eor  #$ff       ; invert
FP 1410          and  #$c0       ; zero others
DE 1420          sta  temp+1
DN 1430          lda  $d018      ; vid.matrix from vic-ii
EF 1440          ror
OF 1450          ror
EM 1460          and  #$3c
NH 1470          ora  temp+1     ; combine
GO 1480          sta  $af
IP 1490  movit   sei
LF 1500          lda  $01
CG 1510          pha
AC 1520          and  #$fd       ; mask out roms
HL 1530          sta  $01
DI 1540          ldx  #$00
LN 1550          ldy  temp
GG 1560          beq  recover
MI 1570          iny             ; #$ff+1=00
GI 1580  store   lda  ($ae),y
LD 1590          sta  ($14),y
AC 1600          iny
BH 1610          bne  store
OB 1620          inc  $15        ; hb dest.
GP 1630          inc  $af        ; hb src.
EE 1640          inx
DB 1650          cpx  #$04
KH 1660          bcc  store
JF 1670  color   lda  #$d8       ; hb of color ram
OK 1680          sta  $af
EM 1690  col1    lda  ($ae),y    ; get a nybble
PE 1700          asl
JF 1710          asl
DG 1720          asl
EL 1730          asl             ; move to hi nybble
FA 1740          sta  temp       ; save it
GN 1750          inc  $af
LB 1760          lda  ($ae),y
NP 1770          and  #$0f
GD 1780          ora  temp
JN 1790          dec  $af
NA 1800          sta  ($14),y
CP 1810          iny
HC 1820          bne  col1
CA 1830          inx
JN 1840          cpx  #$06
GA 1850          beq  done
BP 1860          inc  $15
OE 1870          inc  $af
```

```
IF  1880          inc  $af
NG  1890          bne  col1
CO  1900  ;
CO  1910  recover lda  ($14),y
JP  1920          sta  ($ae),y
KG  1930          iny
NM  1940          bne  recover
DO  1950          inc  $15        ; hb src.
FM  1960          inc  $af        ; hb dest.
OI  1970          inx
NF  1980          cpx  #$04
GN  1990          bcc  recover
KJ  2000  reccol  lda  #$d9       ; pg2 of color ram
IP  2010          sta  $af
HJ  2020  col2    lda  ($14),y    ; get a byte
EN  2030          sta  ($ae),y    ; ignore hi-nybble
BN  2040          lsr
LN  2050          lsr
FO  2060          lsr
MH  2070          lsr             ; move to lo nyb
LP  2080          dec  $af
FC  2090          sta  ($ae),y    ; store it
ED  2100          inc  $af
DK  2110          iny             ; pointer
HF  2120          bne  col2
OC  2130          inx
JP  2140          cpx  #$06       ; enough times
CD  2150          beq  done
NB  2160          inc  $15
KH  2170          inc  $af
EI  2180          inc  $af
NJ  2190          bne  col2
OA  2200  ;
BF  2210  done    pla             ; get config
HE  2220          sta  $01        ; and restore it
EE  2230          cli
JP  2240  endrec  rts
```

Listing 4: 'Move.Plus' in BASIC loader form. Change the value of 'sa' in line 110 to change the location of the routine.

```
FE  100 rem move.plus/c64
GA  110 sa=51200:rem start address
HN  120 ck=0
OE  130 for m=0 to 99:read d
PI  140 poke sa+m,d
KO  150 ck=ck+d: next
AC  160 if ck<>13877 then print"data error!": end
KK  170 print"move.plus installed":print sa "to" sa+110: print
FE  180 print"to use:":print"sys" sa ",source,dest,# of bytes,mask"
BI  1000 data  32, 253, 174,  32, 138, 173,  32, 247
BD  1010 data 183, 132, 195, 133, 196,  32, 253, 174
DD  1020 data  32, 138, 173,  32, 247, 183, 132, 193
BF  1030 data 133, 194,  32, 253, 174,  32, 235, 183
JC  1040 data 138, 240,   8, 120, 165,   1,  72,  41
OB  1050 data 253, 133,   1,  24, 165,  20, 101, 193
IH  1060 data 133,  20, 165,  21, 101, 194, 133,  21
LD  1070 data 160,   0, 177, 195, 145, 193, 230, 195
JB  1080 data 208,   2, 230, 196, 230, 193, 208,   4
OL  1090 data 230, 194, 240,  20, 165, 194, 197,  21
IH  1100 data 208, 232, 165, 193, 197,  20, 208, 226
DJ  1110 data 138, 240,   4, 104, 133,   1,  88,  96
ML  1120 data 136, 208, 245,   0
```

Listing 5: BASIC loader for 'Color.Move'.

```
NK  100 rem color mover
BB  110 sa=51320: rem start address
HN  120 ck=0
GF  130 for m=0 to 127: read d
PI  140 poke sa+m,d
KO  150 ck=ck+d: next
HB  160 if ck<>15890 then print"data error!": end
OB  170 print"color mover installed"
    :print sa "to" sa+127: print
BB  180 print"to store colors:"
    :print"sys" sa ",section # (0-15)": print
KN  190 print"to retrieve colors:"
    : print"sys" sa+4 ",section # (0-15)"
GO  2000 data 160, 255, 208,   2, 160,   0, 132, 195
DI  2010 data  32, 241, 183, 224,  16, 176,  64, 138
BD  2020 data  10,  24, 105, 224, 176,  57, 170, 232
FO  2030 data 240,  53, 162,   0, 133,  37, 134,  34
EH  2040 data 134,  36, 162, 254, 164, 195, 240,  42
PJ  2050 data 200, 169, 216, 133,  35, 177,  34,  10
CP  2060 data  10,  10,  10, 133, 195, 230,  35, 177
OI  2070 data  34,  41,  15,   5, 195, 198,  35, 145
GF  2080 data  36, 200, 208, 233, 232, 240,  55, 230
IE  2090 data  37, 230,  35, 230,  35, 208, 222,  76
FH  2100 data  72, 178, 120, 165,   1,  72,  41, 253
CE  2110 data 133,   1, 169, 217, 133,  35, 177,  36
HN  2120 data 145,  34,  74,  74,  74,  74, 198,  35
IF  2130 data 145,  34, 230,  35, 200, 208, 239, 232
ED  2140 data 240,   8, 230,  37, 230,  35, 230,  35
KH  2150 data 208, 228, 104, 133,   1,  88,  96,   0
```

Listing 6: BASIC loader for 'Video.Move'.

```
JK  100 rem video mover
GC  110 sa=51456: rem start address
HN  120 ck=0
NE  130 for m=0 to 203: read d
PI  140 poke sa+m,d
KO  150 ck=ck+d: next
MC  160 if ck<>28129 then print"data error!": end
BC  170 print"video mover installed"
    : print sa "to" sa+203: print
AJ  180 print"to store video:"
AP  190 print"sys" sa ",section # (0-4), screen org": print
CM  200 print"to retrieve video:"
PM  210 print"sys" sa+4 ",section # (0-4), screen org"
OM  3000 data 160, 255, 208,   2, 160,   0, 132, 195
NA  3010 data  32, 241, 183, 224,   5, 176,  17, 169
NP  3020 data 224, 202,  48,   7,  24, 105,   6, 176
BP  3030 data   7, 208, 246, 168, 105,   5, 144,   3
DM  3040 data  76,  72, 178, 132,  21, 173, 136,   2
PE  3050 data 133, 175, 169,   0, 133, 174, 133,  20
BD  3060 data  32, 241, 183, 138, 240,  23, 173,   0
CB  3070 data 221, 106, 106, 106,  73, 255,  41, 192
AH  3080 data 133, 196, 173,  24, 208, 106, 106,  41
BN  3090 data  60,   5, 196, 133, 175, 120, 165,   1
CC  3100 data  72,  41, 253, 133,   1, 162,   0, 164
JK  3110 data 195, 240,  57, 200, 177, 174, 145,  20
GB  3120 data 200, 208, 249, 230,  21, 230, 175, 232
BD  3130 data 224,   4, 144, 240, 169, 216, 133, 175
LF  3140 data 177, 174,  10,  10,  10,  10, 133, 195
JK  3150 data 230, 175, 177, 174,  41,  15,   5, 195
NG  3160 data 198, 175, 145,  20, 200, 208, 233, 232
FG  3170 data 224,   6, 240,  58, 230,  21, 230, 175
JH  3180 data 230, 175, 208, 220, 177,  20, 145, 174
MF  3190 data 200, 208, 249, 230,  21, 230, 175, 232
KH  3200 data 224,   4, 144, 240, 169, 217, 133, 175
LM  3210 data 177,  20, 145, 174,  74,  74,  74,  74
KN  3220 data 198, 175, 145, 174, 230, 175, 200, 208
NG  3230 data 239, 232, 224,   6, 240,   8, 230,  21
NL  3240 data 230, 175, 230, 175, 208, 226, 104, 133
JC  3250 data   1,  88,  96,   0
```

News BRK

Transactor News

Submitting News BRK Press Releases

If you have a press release you would like to submit for the *News BRK* column, make sure that the computer or device for which the product is intended is prominently noted. We receive hundreds of press releases for each issue and ones whose intended readership is not clear must unfortunately go straight into the trash bin. It should also be mentioned here that we only print product releases which are in some way applicable to Commodore equipment. News of events such as computer shows should be received at least 6 months in advance. The *News BRK* column is compiled solely from press releases and is intended only to disseminate information; we have not necessarily tested the products mentioned. Items for publication should be sent to Moya Drummond along with any queries about advertising, ads themselves and editorial queries.

Demand for T-Shirts Outstrips Supply

We apologize to all those readers who sent in for T-Shirts and whom we have had to disappoint. Demand was such that our initial supply was sold out within a week and we are having trouble finding another supplier. Most of you should have received a letter from our new Customer Service Assistant, Renanne Turner, offering a refund in the case of T-Shirts which were paid for in advance, or a Potpourri or TransBASIC II disk for subscribers who were entitled to a freebie for ordering both the *Transactor* Magazine and Disk at the same time.

We have had a similar problem with the G-Link IEEE interface. However, we now have a fresh supply and are shipping these again. If you are one of the customers who has been waiting patiently, please bear with us just a little longer.

New Look For Customer Service

Jennifer has moved on to new pastures and we now have Renanne Turner doing her utmost to create order from chaos in the field of subscriptions and orders. It is a huge job and growing ever more complex now that we have two magazines and both are expanding rapidly. Renanne is working miracles and on Mondays, Wednesdays and Fridays will be delighted to resolve your queries. On Tuesdays and Thursdays, however, she has to have time to keep her database and mail order departments up to date and, therefore, the phones will be answered by a machine. We think you will find that from now on delays and mistakes will be kept to a minimum and we hope you will appreciate the new system.

Subscription Switch

Remember that if you wish to switch your subscription to *Transactor for the Amiga* there is no charge. However, there does seem to have been some confusion: you can only switch the number of issues of *Transactor* Classic to which you are still entitled. TPUG subscribers please remember that your subscription is with TPUG, not directly with *Transactor,* and we are therefore unable to switch issues due to you from them. Please be sure to put your name and ZIP or Postal Code as well as your subscriber number on your order card.

The 20/20 Deal

...is still in effect: order 20 subscriptions to the mag or disk, 20 back issues, 20 disks etc., and get a 20% discount. (Offer applies to regular prices and cannot be combined with other specials.)

No Longer Available

As mentioned in the last issue, the 1541 Upgrade ROM Kit is now discontinued. Please see Vol.7 Issue 2 for complete instructions on obtaining a set; disk #13 contains the ROM image you'll need to burn your own EPROMS. However, we're reasonably sure that the ROM image is compatible with the 1541 *only*. 1541**C** owners will need to create an image of their ROM set, then make the changes described in V7 I2, but with minor adjustments to accommodate for what are more than likely simple address changes. We are still waiting for an update article from someone who has successfully done this!

'Moving Pictures' is also out of stock and no longer available from Transactor. If you have ordered a copy, you may ask either for a refund or have a credit issued against further orders from Transactor Publishing - Renanne will be in touch with you. Moving Pictures is now being distributed by CDA, with new packaging and manual. Contact CDA at: P.O. Box 1052, Yreka, CA 96097. Phone (916) 842-3431.

Transactor Mail Order

It is perhaps worth mentioning that items on order cards in back issues of the *Transactor* are not necessarily currently available; if you are unsure, please call Renanne before sending in your order. To be certain, place orders from the card in the most recent issue.

Prices for all products are listed on the order card in the centre of the magazine. Subscribers: you can use the address label from the bag holding your magazine and just stick it on the order card instead of filling it in by hand!

• **Quick Brown Box** - Battery Backed RAM for C64 or C128. The Quick Brown Box cartridges for the C64 and 128

can be used to store any type of programs or data that remains intact even when the cartridge is unplugged. Unlike EPROM cartridges, the QBB requires no programming or erasing equipment except your computer. Loader programs are supplied and you can store as many programs into the cartridge as its memory will allow. It may even be used as a non-volatile RAM disk. Auto-start programs are supported such as BBS programs and software monitoring systems that need to re-boot themselves in the event of a power failure. All models come with a RESET push button and use low current CMOS RAM powered by a 160 mA-Hr. Lithium cell with an estimated life of 7 to 10 years. Comes with manual; software supplied includes loader utilities and Supermom+64 (by permission of Jim Butterfield). 30 day money back guarantee and a 1 year repair/replacement warranty.

• **The Potpourri Disk** - A C64 product from the software company AHA! (aka Chris Zamara and Nick Sullivan). Includes a wide assortment of 18 programs ranging from games to educational programs to utilities. All programs can be accessed from a main menu or loaded separately. No copy protection is used on the disk so you can copy the programs you want to your other disks for easy access. Built-in help is available from any program at any time with the touch of a key, so you never need to pick up a manual or exit a program to learn how to use it. Many of the programs on the disk are of a high enough quality that they could be released on their own, but you get all 18 on the Potpourri disk for just $17.95 US/$19.95 Canadian.

• **TransBASIC II** - TransBASIC II contains all TB modules ever printed. There are over 140 commands at your disposal; you pick the ones you want to use in any combination. It's so simple that a summary of instructions fits right on the disk label. The manual describes each of the commands, plus how to write your own commands. People who ordered TB1 can upgrade to TBII for the price of a regular Transactor disk (8.95/9.95). If you are upgrading, please let us know on the order form.

• **Inner Space Anthology** - This is our ever-popular reference book. It has no "reading" material, but in 122 compact pages there are memory maps for five CBM computers, three disk drives and maps of COMAL; summaries of BASIC commands, Assembler and MLM commands and Wordprocessor and Spreadsheet commands. ML codes and modes are summarized, as well as entry points to ROM routines. There are sections on Music, Graphics, Network and BBS phone numbers, Computer Clubs, Hardware, unit-to-unit conversions, plus much more ... about 2.5 million characters in total!

• **The Transactor Bits and Pieces Book and Disk** - 246 pages of Bits from *Transactor* Volumes 4 through 6 with a very comprehensive index. Even if you have all those issues, it makes a handy reference - no more flipping through magazines for that one bit that you just know is somewhere ... Also, each item if forward/reverse referenced. Occasionally the items in the Bits column appeared as updates to previous bits. Bits that were similar in nature are also cross-referenced. And the index makes it even easier to find those quick tips that eliminate a lot of wheel re-inventing. The bits book disk contains all programs from the book and can save a lot of typing.

• **The G-Link Interface** - The G-Link is a Commodore 64 to IEEE interface. It allows the 64 to use IEEE peripherals such as the 4040, 8050, 9090, 9060, 2031 and SFD-1001 disk drives, or any IEEE printer, modem or even some Hewlett-Packard and Tektronics equipment like oscilloscopes and spectrum analyzers. The beauty of the G-Link is its "transparency" to the C64 operating system. Some IEEE interfaces for the 64 add BASIC 4.0 commands and other things to the system that can interfere with utilities you might like to install. The G-Link adds nothing: it is so transparent that a switch is used to toggle between serial and IEEE modes, not a linked-in command. Switching from one mode to the other is also possible with a small software routine as described in the documentation.

• **Transactor Disks** - now with their new, colour directory listing labels. As of Disk #19 a modified version of Jim Butterfield's *Copy-All* will be on every disk. It allows file copying from serial to IEEE drives, or vice versa.

• **The Micro-Sleuth:** C64/1541 Test Cartridge - Designed by Brian Steele (a service technician for several southern Ontario Schools), this is a very popular cartridge. The Micro-Sleuth will test the RAM of a C64 even if the machine is too sick to run a program! The cartridge takes complete control of the machine, tests all RAM, ROM and other chips, and in another mode puts up a menu:

1) Check drive speed	5) Joystick port 1 test
2) Check drive alignment	6) Joystick port 2 test
3) 1541 serial test	7) Cassette port test
4) C64 serial test	8) User port test

A second board (included) plugs onto the User Port: it contains 8 LEDs that let you zero in on the faulty chip. Complete with manual.

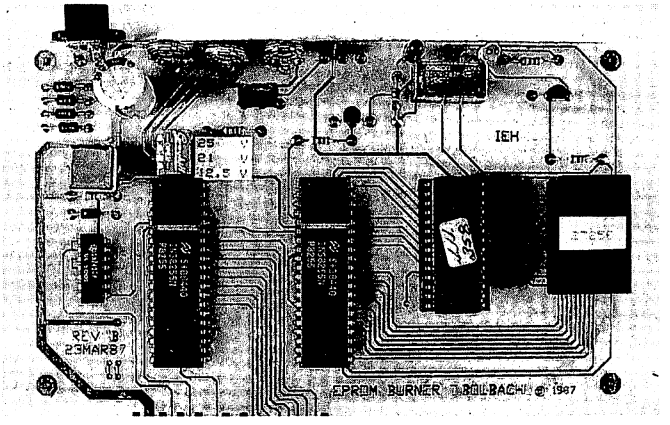
• **Transactor Back Issues and Microfiche** - All Transactors from Volume 4 Issue 1 are available on Microfiche. The strips are the 98 page size compatible with most fiche readers. Some issues are available *only* on microfiche and are marked as such on the order card. The price is the same as for the magazines with the exception that a complete set (Volumes 4, 5, 6 and 7) will cost just $49.95 US/$59.95 CDN.

This list shows the "themes" of each issue. Theme issues didn't start until Volume 5 Issue 1. Transactor Disk #1 includes all the programs from Volume 4 and Disk #2 includes all programs for Volume 5 Issues 1 to 3. Thereafter there is a separate disk for each issue. Disk #8 from the Languages Issue includes COMAL 0.14, a soft-loaded, slightly scaled down version of the COMAL 2.0 cartridge. Volume 6, Issue 5 lists the directories for Transactor Disks #1 to #9.

- Vol.4 Issues 1 to 3 (Disk #1)
- Vol.4 Issues 4 to 6 (Disk #1) - MF only
- Vol.5 Issue 1 - Sound and Graphics Disk # 2
- Vol.5 Issue 2 - Transition to ML (MF only) # 2
- Vol.5 Issue 3 - Piracy and Protection (MF only) # 2
- Vol.5 Issue 4 - Business and Eduction (MF only) # 3
- Vol.5 Issue 5 - Hardware and Peripherals # 4
- Vol.5 Issue 6 - Aids & Utilities # 5
- Vol.6 Issue 1 - More Aids & Utilities # 6
- Vol.6 Issue 2 - Networking & Communications # 7
- Vol.6 Issue 3 - The Languages # 8
- Vol.6 Issue 4 - Implementing the Sciences # 9
- Vol.6 Issue 5 - Hardware & Software Interfacing #10
- Vol.6 Issue 6 - Real Life Applications #11
- Vol.7 Issue 1 - ROM/Kernel Routines #12
- Vol.7 Issue 2 - Games from the Inside Out #13
- Vol.7 Issue 3 - Programming the Chips #14
- Vol.7 Issue 4 - Gizmos and Gadgets #15
- Vol.7 Issue 5 - Languages II #16
- Vol.7 Issue 6 - Simulations & Modelling #17
- Vol.8 Issue 1 - Mathematics #18
- Vol.8 Issue 2 - Operating Systems #19
- Vol.8 Issue 3 - Feature: Surge Protector #20
- Vol.8 Issue 4 - Feature: Transactor for the Amiga #21
- Vol.8 Issue 5 - Feature: Binary Trees #22

Industry News

EPROM Programmer for the C64, C128 and 64C from B & B Products is a versatile programmer offered in an easy-to-assemble kit form or as a complete and tested system. The design is based on the article by T. Bolbach featured in the January 1987 issue of Transactor and contains many enhancements, including on-board selectable programming voltages, local reset switch, power transformer and super-fast improved software. The programmer supports 2716 through 27256-type EPROMs and also programs the 68764 direct replacement types for the Kernal and BASIC ROMs. Documentation includes the schematic. Complete kit with all parts $59.00 (US). Completed and tested units, $89.00 (US). Send cheque or money order to: T. Bolbach, 1575 Crestwood, Toledo, OH 43612.



The Super Chips: New from Free Spirit Inc. is a custom operating system for the Commodore 128. The system consists of three 16K chips labelled Basic Lo, Basic Hi and Kernal which replace U33, U34 and U35 on the motherboard of the C128. The Super Chips add a variety of powerful new commands and functions to the C128 operating system including: Type, (Restore) D, Combine, Merge, File, Change, Find, * - Send monitor command to the printer, and Editor. When done, a Basic program can be compiled which can be incorporated into a program and/or saved to disk. The custom operating system also redefines the function keys.

In 80 column mode the F3/F4 keys will simultaneously display the directories from devices 8 and 9 in separate windows on the screen. The operating system will default to fast mode when powered up or reset with the 40/80 display button down. It will default to slow in the 40 column mode.

The Super Chips system is compatible with 1541/1571/1581 disk drives and virtually all Commodore software and peripherals. Similar systems will be available for the 128D and C64 in the near future.

Available at $49.95 from Joe Hubbard, Free Spirit Software Inc., 905 W. Hillgrove, Suite 6, La Grange, IL 60525 (312)352-7323.

New Basic for GEOS: BeckerBASIC adds more than 270 new commands and functions to the Commodore 64 and GEOS. It has commands for screen and cursor control, hi-res graphics and sprite animation, sound and music, structured programming and programmers' aids. A program written in BeckerBASIC runs as a GEOS application and can use GEOS' pull-down menus, dialog boxes, different fonts, hi-res graphics and fill patterns and more.

BeckerBASIC can be customized by adding user-defined commands and function key definitions. The BeckerBASIC package includes a free run-time version so that BeckerBASIC applications may be distributed to other GEOS users. BeckerBASIC is compatible with Commodore 64 BASIC and GEOS Version 1.3. The suggested retail price of BeckerBASIC is $49.95.

Available from any Abacus dealer or distributor or call (616) 698-0330. Abacus is at 5370 52nd Street SE, Grand Rapids, MI., USA 49508.

Science Software is a series of tutorial, utility and application computer programs in the areas of Astronomy, Earth Satellites and Aeronautics. The Astronomy disk contains programs for determining the position of the Sun, Moon, Planets and Stars. The Earth Satellite programs can be used to determine the location of TVRO, weather, OSCAR and other earth satellites. The Aeronautics disk contains programs for model rocketry, hot air balloons and gliders. Science Software disks are available for the Commodore 64/128 (in C64 mode) and the Amiga. For additional information contact David Eagle, Sci-

ence Software, 7370 S. Jay Street, Littleton, CO 80123 (303) 972-4020.

Poseidon Electronics announces an addendum to its catalog of disks for Commodore 64/128 CP/M users. The full catalog is available for $4.10 plus $0.90 SASE (please send a *large* (envelope) in limited quantities; the addendum costs $1.60 plus a $0.39 SASE. Poseidon have a large library of CP/M disks; for further information contact Ralph S. Lees Jr., Poseidon Electronics, 103 Waverley Place, New York, NY 10011 (212) 777-9515.

RS-232 interface for the C64, C128 and Vic 20: Now you can connect a true RS-232 modem or printer to your 64 or 128. This interface supports all the RS-232 control lines (DTR, RTS, SI, RI, CTS, DSR and DCD). Also, if you are using a C128, the interface does not obstruct the 80 column video port. It uses only the +5 volt line from the computer and draws 30 milliamps of current. It is packaged as a 3 inch square PC board and terminated with a female DB-25 connector, at a cost of $55 (CDN).

PET to Centronics interface: supports all standard Centronics style printers and is small enough to be left inside the PET itself jumpered to the IEEE connector at the rear of the computer. Costs $50 (CDN).

Both interfaces come with a 90 day warranty. For further information and orders contact Chris Czech, 227-7a Street, NE., Calgary, Alberta, Canada, T2E 4E7. (403) 262-3587

Surge & Lightning Protection for datacommunications and computer interfaces described from analysis to solution. Telebyte Technology's expanded line of surge and lightning protection products is described in a six-page brochure together with explanations of the phenomena and the basic techniques for protection. A selection chart is included to simplify the process of choosing the best device and custom products are available. For further information contact Telebyte Technology Inc., 270 E. Pulaski Road, Greenlawn, NY., USA. 11740 (516) 423-3232 or (800) 835-3298.

The Strategist for the Commodore 128 is a market timing program for investors in stocks, bonds, mutual funds and commodities. It allows the user to plot prices on the same chart as one or several market indicators so that he can pick the ones he wants to time his trades. There is an historical file which makes realistic simulated trades to see how a strategy would have paid off in real life; it then repeats this, varying the strategy each time until it arrives at the one which gives the highest payoff. The system uses high-low trading enhanced by the use of persistence checks to confirm buy and sell signals and an exponential moving average of quote to quote volatility.

The Strategist master disk, an 80 page manual and 90 day money-back guarantee cost $29.95. The system is copyrighted and is distributed on a shareware basis. Purchasers may give, but not sell, copies to other users to try; if they like the program, those with try-out copies should send $29.95 to Strategy Software in order to become registered users and obtain the latest version of the program and manual. For Commodore 128 users with a 1541 or 1571 disk drive; a printer is desirable but not essential. Dealer inquiries are welcome. A C64 version will soon be available at $24.95. Contact Strategy Software, 909 Carol Lane, Fairbanks, AK, USA, 99712 (907) 457-2294.

New Telecommunications Software from Made in America:

BananaTerm! offers all the standard features expected in a terminal program along with an advanced phone book, support for up to 600 baud on standard 300 baud modems and custom character graphics.

System 64! is a bulletin board system designed to run on standard C64/128 systems. It creates an environment enabling communication between computers and modems for the exchange of files and messages. It features ALEX programming language, customizable environment and support for up to 600 baud on standard 300 baud modems.

The two systems are complementary but are available separately and are compatible with other telecommunications environments. BananaTerm! costs $24.95 and System 64! costs $49.95 from Made in America, 9069 Sussex, Union Lake, MI., USA. 48085 (313) 698-2104.

RomJet Custom Cartridges: 32K TO 256K RomJet cartridges for C64 and C128 modes are here. These cartridges let you access your favourite software instantly, via menus that are also in the cartridge.

RomJet will install on its cartridges any Basic, compiled, or machine language non-copy-protected program, including programs such as Paperclip, Consultant, and WordPro, for which the copyright states that you, the proven legal purchaser, can make a back-up copy for your own personal use. You must present to RomJet your original purchase receipt, or proof of purchase seal.

Commercially sold programs belonging to companies or authors other than RomJet that have copy-protection and/or for which the copyright states that you are *not* allowed to make a back-up copy for your own personal use cannot be put on a RomJet cartridge.

Prices range from $32.00 (CDN) for 8K up to $196 for 256K cartridges. Cartridges may be upgraded to the next larger size for the difference in price. There are extra charges for cartridges that require programmer's time (e.g. modifying a program that loads files from disk to work from the cartridge exclusively).

Inquire at: RomJet, 210-2450 Sheppard Ave. E. Willowdale, Ontario M2J 4Z9. Phone (416) 274-7378 or 626-5959.

The Potpourri Disk

Help!

This HELPful utility gives you instant menu-driven access to text files at the touch of a key – while any program is running!

Loan Helper

How much is that loan really going to cost you? Which interest rate can you afford? With Loan Helper, the answers are as close as your friendly 64!

Keyboard

Learning how to play the piano? This handy educational program makes it easy and fun to learn the notes on the keyboard.

Filedump

Examine your disk files FAST with this machine language utility. Handles six formats, including hex, decimal, CBM and true ASCII, WordPro and SpeedScript.

Anagrams

Anagrams lets you unscramble words for crossword puzzles and the like. The program uses a recursive ML subroutine for maximum speed and efficiency.

Life

A FAST machine language version of mathematician John Horton Conway's classic simulation. Set up your own 'colonies' and watch them grow!

War Balloons

Shoot down those evil Nazi War Balloons with your handy Acme Cannon! Don't let them get away!

Von Googol

At last! The mad philosopher, Helga von Googol, brings her own brand of wisdom to the small screen! If this is 'AI', then it just ain't natural!

News

Save the money you spend on those supermarket tabloids – this program will generate equally convincing headline copy – for free!

Wrd

The ultimate in easy-to-use data base programs. WRD lets you quickly and simply create, examine and edit just about any data. Comes with sample file.

Quiz

Trivia fanatics and students alike will have fun with this program, which gives you multiple choice tests on material you have entered with the WRD program.

AHA! Lander

AHA!'s great lunar lander program. Use either joystick or keyboard to compete against yourself or up to 8 other players. Watch out for space mines!

Bag the Elves

A cute little arcade-style game; capture the elves in the bag as quickly as you can – but don't get the good elf!

Blackjack

The most flexible blackjack simulation you'll find anywhere. Set up your favourite rule variations for doubling, surrendering and splitting the deck.

File Compare

Which of those two files you just created is the most recent version? With this great utility you'll never be left wondering.

Ghoul Dogs

Arcade maniacs look out! You'll need all your dexterity to handle this wicked joystick-buster! These mad dog-monsters from space are not for novices!

Octagons

Just the thing for you Mensa types. Octagons is a challenging puzzle of the mind. Four levels of play, and a tough 'memory' variation for real experts!

Backstreets

A nifty arcade game, 100% machine language, that helps you learn the typewriter keyboard while you play! Unlike any typing program you've seen!

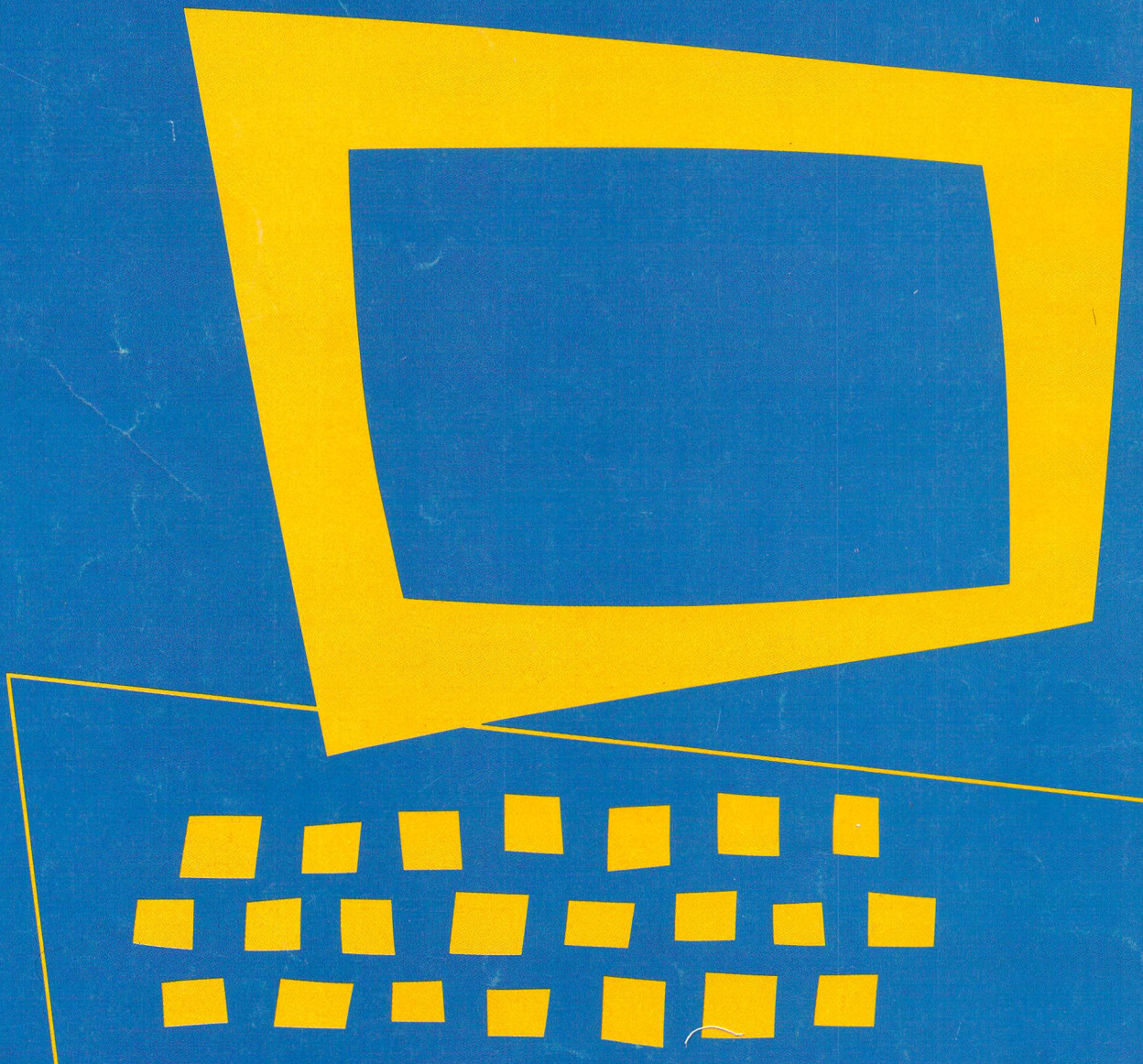
All the above programs, just $17.95 US, $19.95 Canadian. No, not EACH of the above programs, ALL of the above programs, on a single disk, accessed independently or from a menu, with built-in menu-driven help and fast-loader.

The ENTIRE POTPOURRI COLLECTION JUST $17.95 US!!

See Order Card at Center

Computer Expo

April 14, 15, 16, 17, 1988
Toronto International Centre



THE MICROCOMPUTER SHOW FOR EVERYONE

Produced by The Hunter Group Inc. (416)595-5906