CDC 00594

# The Transactor

The Tech/News Journal For Commodore Computers

95% Advertising Free! Nov. 1985: Volume 6, Issue 03. $2.95

## The Languages

- COMAL Tips and Procedure Reference
- LOGO Commands and Statements
- Jumbo TransBASIC
- Instruction Set Reference: 6809, 68000, 8088
- A PC Summary Reference: BASIC and MS DOS
- Butterfield: Commodore 128 Keywords and Tokens
- Scaled High-Res Text Generator
- Comparing Applesoft BASIC with Commodore 2.0
- Hidden 6502/6510 Op-Codes
- A BASIC Execution Speed Analyzer
- Plus: A look at the New Atari 520 ST
- Commodore 64 LIST Freeze
- Still More about the SAVE@ Dilemma

HIGH QUALITY LOW PRICE

COMMODORE 64® SOFTWARE

# The GOLD DISK

### VOLUME 1 FEATURES:
- **Complete Data Base system** with:
  - — Full screen data entry
  - — Fast sorting capabilities
  - — Extensive multi-clause search facilities
  - — On line help facilities
  - — On disk tutorial and instructions
- **2 arcade style games** with 3D graphics
- **Learning to Program in BASIC:** Lesson 1 Input/output, variables and assignments, conditional branches.
- **Towers of Hanoi** with solution
- **The Personal House Purchase Analyzer:** Evaluation of financial scenarios

### VOLUME 2 FEATURES:
- **Complete Word Processing Package** with:
  - — Cut-and-paste, full formatting capabilities, Margin Justification, Page Numbering
  - — Access to programmable printer capabilities
- **4 action screens game! "SPOOK CASTLE"!**
- **Children's educational game:** counting
- **Learning to Program in BASIC** — Lesson 2: Array & string variables, FOR-NEXT loop
- **8 Queen's puzzle** with solution
- **Loans and Mortgages program**
- **File backup/copy program utility**

### VOLUME 3 FEATURES:
- **A complete Typing Tutor program** Use the built in lessons or create your own! Complete with self evaluator
- **Arcade action game** "Chomper"
  - — 4 screens to complete at each level
- **Mind puzzling strategy game:** "Cubic"
- **Home Budgetter program**
- **Report Generator** (used with Vol. 1 database)
- **Tutorial:** Powerful sorting techniques
- **Learning to Program in Basic:** Lesson 3: PEEKs, POKES, print controls, multi-dimensional arrays and subroutines

### VOLUME 4 FEATURES:
- **The Gold Disk Art package** High-res joystick controlled painter prgm with these powerful features:
  - — Preprogrammed shapes
  - — 'Fill' capabilities, Free drawing mode
  - — 'Undo' capability for error-correction
  - — 'Print' capability, Zoom
- **The Gold Disk Trivia Game** with
  - — Over 500 questions, Multi player, scoring
- **Arcade action game** with 3D graphics
- **Educational:** File operations on the C64
- **Programming Tutorial:** Binary Search technique illustrated
- **Complete checkwriter program**

**"I've always thought good programs cost a lot more. Now there's Gold Disk."** Karl Hildon, TRANSACTOR

**Official approval from "Small Things Considered"** Radio WNYC AM83, New York

**"A great way to get inexpensive quality software"** L. Redgers, TPUG Magazine

**"The products low price, high quality and simplicity of use, make it an excellent addition . . ."** CLASSIC Bookshops

**"Gold Disk remains on top."** DASHER INDUSTRIES, Distributor

**". . . the right kind of product at the right prices."** COLES, THE BOOK PEOPLE

**"Gold Disk idea's time has come."** Phil Kemp, INPUT Magazine

### VOLUME 5 FEATURES:
- **The Gold Disk Sprite and Font Editor package** Joystick controlled graphic package for Sprites and character graphics with:
  - — Multi-colour capabilities
  - — High resulution capabilities
  - — Easy to follow instructions
- **Challenge game:** "Kode Breaker"
- **Arcade action, arcade graphics game:** "Cosmic Chaos" — Outstanding play!!
- **Educational Tutorial:** How to Use Sprites
- **"Knights Tour"** with solution

### VOLUME 6 FEATURES
- **Enhanced Basic**
  - — Over fifty powerful new commands and functions
  - — Increases programmer productivity
  - — Simplifies programming
  - — Easy-to-use
- **Action Strategy Game** "Rogi Ginbani"
- **4 Screen Game:** "Return to Spook Castle"
- **Sound Tutorial** — Part I
- **Tutorial** on "Disk Manipulation"
- **Educational:** Encoding — Decoding
- **Financial Options Program** — Evaluate different payment scenarios

### VOLUME 7 FEATURES:
- **Full Spreadsheet Program**
  - — Ability to evaluate "What if" scenarios
  - — Powerful and easy-to-use tool with
  - — Large Worksheet, Variable width columns
  - — Internal File Operations, Printer Option
- **Casino Pack** "Poker" and "Black Jack"
- **Bonus Card Game** "Solitaire"
- **Sound Tutorial** — Part II
- **Tutorial** on error recovery
- **Educational:** Stacks
- **Business:** Balance Sheet

### VOLUME 8 FEATURES:
- **FORTH!** The ease of BASIC, the Power of Machine Code Complete implementation of FIG FORTH!
- **The Gold Disk Trivia Game II**
  - — Over 500 new and interesting questions
  - — Multi-player, auto-scoring
- **Arcade action game** "Cube Slinker" 3D action, multi-level
- **Programming with Forth** Part I
- **Tutorial** — Screen manipulation
- **Educational** — working with queues
- **Business:** Compound and Simple Interest

The GOLD DISK

2179 Dunwin Drive, #6, Mississauga
Ontario, Canada L5L 1X2
or
P.O. Box 1540, Buffalo
N.Y. 14205 - 1540

PHONE ORDERS
**(416) 828 0914**

**PRICES**

| | |
|---|---|
| SINGLE ISSUE | @ 15.95 (U.S. FUNDS) OR 17.95 (CDN. FUNDS) |
| 3 ISSUES | @ 39.95 (U.S. FUNDS) OR 45.95 (CDN. FUNDS) |
| 6 ISSUES | @ 60.95 (U.S. FUNDS) OR 75.95 (CDN. FUNDS) |
| 8 ISSUES | @ 80.95 (U.S. FUNDS) OR 99.95 (CDN. FUNDS) |

Ontario residents add 7% tax. Allow up to 4 weeks for delivery. Payment must accompany all orders.

Volume 6
Issue 03
Circulation 64,000

The Transactor

# The Languages Issue



**Note: Before entering programs, see "Verifizer" on page 4**

The Transactor
The Tech/News Journal For Commodore Computers

**Managing Editor**
Karl J. H. Hildon

**Editor**
Richard Evers

**Technical Editor**
Chris Zamara

**Art Director**
John Mostacci

**Administration & Subscriptions**
Lana Humphries

**Contributing Writers**
Robert Adler
Gary Anderson
M. Van Bodegom
Peter Boisvert
Anthony Bryant
Tim Buist
Jim Butterfield
Gary Cobb
Bob Davis
Elizabeth Deal
Yijun Ding
Tony Doty
Michael J. Erskine
Jeff Goebel
Jim Grubbs
Gary Gunderson
Dave Gzik
Bob Hayes
Thomas Henry
Chris Johnsen
Steve Kortendick
Scott Maclean
Jim McLaughlin
John R. Menke
Gerald Neufeld
John W. Ross
Louis F. Sander
Darren J. Spruyt
Aubrey Stanley
David Stevenson
Howard Strasberg
Nick Sullivan
Tony Valeri
Charles Whittern

**Production**
Attic Typesetting Ltd.

**Printing**
Printed in Canada by
MacLean Hunter Printing

## Program Listings In The Transactor

All programs listed in The Transactor will appear as they would on your screen in Upper/Lower case mode. To clarify two potential character mix-ups, zeroes will appear as '0' and the letter "o" will of course be in lower case. Secondly, the lower case L ('l') has a flat top as opposed to the number 1 which has an angled top.

Many programs will contain reverse video characters that represent cursor movements, colours, or function keys. These will also be shown exactly as they would appear on your screen, but they're listed here for reference. Also remember: CTRL-q within quotes is identical to a Cursor Down, et al.

Occasionally programs will contain lines that show consecutive spaces. Often the number of spaces you insert will not be critical to correct operation of the program. When it is, the required number of spaces will be shown. For example:

print "          flush right " – would be shown as – print "[10 spaces]flush right "

### Cursor Characters For PET / CBM / VIC / 64

| | | | |
|---|---|---|---|
| **Down** | q | **Insert** | T |
| **Up** | Q | **Delete** | t |
| **Right** | ] | **Clear Scrn** | S |
| **Left** | [Lft] | **Home** | s |
| **RVS** | r | **STOP** | c |
| **RVS Off** | R | | |

### Colour Characters For VIC / 64

| | | | |
|---|---|---|---|
| **Black** | P | **Orange** | A |
| **White** | e | **Brown** | U |
| **Red** | £ | **Lt. Red** | V |
| **Cyan** | [Cyn] | **Grey 1** | W |
| **Purple** | [Pur] | **Grey 2** | X |
| **Green** | ↑ | **Lt. Green** | Y |
| **Blue** | ← | **Lt. Blue** | Z |
| **Yellow** | [Yel] | **Grey 3** | [Gr3] |

### Function Keys For VIC / 64

| | | | |
|---|---|---|---|
| **F1** | E | **F5** | G |
| **F2** | I | **F6** | K |
| **F3** | F | **F7** | H |
| **F4** | J | **F8** | L |

**Please Note: The Transactor has a new phone number: (416) 878 8438**

The Transactor is published bi-monthly by Transactor Publishing Inc., 500 Steeles Avenue, Milton, Ontario, L9T 3P7. Canadian Second Class mail registration number **6342.** USPS **725-050,** Second Class postage paid at Buffalo, NY, for U.S. subscribers. U.S. Postmasters: send address changes to The Transactor, 277 Linwood Avenue, Buffalo, NY, 14209 **ISSN# 0827-2530.**

The Transactor is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names (PET, CBM, VIC, 64) are registered trademarks of Commodore Inc.

Subscriptions:
Canada $15 Cdn. U.S.A. $15 US. All other $21 US.
Air Mail (Overseas only) $40 US. ($4.15 postage/issue)

**Send all subscriptions to:** The Transactor, Subscriptions Department, 500 Steeles Avenue, Milton, Ontario, Canada, L9T 3P7, 416 878 8438. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to Milton HQ. For best results, use postage paid card at center of magazine.

Back Issues: $4.50 each. Order all back issues from Milton HQ.

**SOLD OUT:** The Best of The Transactor Volumes 1 & 2 & 3; Vol 4 Issues 04, 05, 06, Vol 5 Issues 03, 04
**Still Available:** Vol. 4: 01, 02, 03. Vol. 5: 01, 02, 04, 05, 06. Vol. 6: 01, 02, 03

**Quantity Orders:**

CompuLit
PO Box 352
Port Coquitlam, BC
V5C 4K6
604 941 7911

Micron Distributing
409 Queen Street West
Toronto, Ontario, M5V 2A5
(416) 593 9862
Dealer Inquiries ONLY:
1 800 268 9052
Subscription related inquiries
are handled ONLY at Milton HQ

U.S.A. Distributor:

Capital Distributing
Charlton Building
Derby, CT
06418
(203) 735 3381
(or your local wholesaler)

Master Media
261 Wyecroft Road
Oakville, Ontario
L6J 5B4
(416) 842 1555
(or your local wholesaler)

Editorial contributions are always welcome. Writers are encouraged to prepare material according to themes as shown in Editorial Schedule (see list near the end of this issue). Remuneration is $40 per printed page. Preferred media is 1541, 2031, 4040, 8050, or 8250 diskettes with WordPro, WordCraft, Superscript, or SEQ text files. Program listings over 20 lines should be provided on disk or tape. Manuscripts should be typewritten, double spaced, with special characters or formats clearly marked. Photos or illustrations will be included with articles depending on quality. Authors submitting diskettes will receive the Transactor Disk for the issue containing their contribution.

# Start Address

Two things I'd like to say. Both semi-related. Here it is.

Two barrier zones have developed on either side of our fair industry, leaving us in the middle. And when I say *us*, I mean you, and I. Because we are those with that affinity for our micros, much like musicians have for their instruments. And much like an instrument, each zone has several sounds, but has the same tone throughout the entire range.

In one zone, perhaps to our far left, is the general conception that the microcomputer is over. A new toy, a new hobby, a passing fad not unlike any other. They'll say, "get out now before it's too late" to the retailer sporting a blank purchase order. Quite contrary to the "ya, let's sell micros" attitude of only a short while ago.

Towards the inside of the same zone is the opinion that the seige is over, and the micro necessity will be determined by the individual – not the public at large whence many took the plunge only because the Jones's did. They'll say, "micros will continue to sell just like guitars will, but I'm not sure *I* really want one, and I've no plans to start selling them either. I want something different because I want to be there as it happens, as opposed to having a lot of catching up to do".

Then there's us, once again, who know we're in love, who know there will be those who resist and chastise, who know there will be those without the will to participate in our domain which they would probably find fascinating given half a chance. But we also know that new interest is being generated, that new faces will indeed make their entrance, and names among those faces will make their presence known. The names and faces lie in the zone to our right, which for the most part will always be just beyond the horizon.

To our inside right are those who believe, "the micro is for me". They've decided that a micro would be a fabulous pastime, a challenge, and a chance to learn something which just might have alterior benefits one day. They may not know quite yet which brand to buy, but they will. And when they buy, it won't be long until many are among us. They may not buy Commodore, but nonetheless they will want to advance and meet others who enjoy the same stimulus.

Shortly beyond here lies the average. Parents advocate microcomputing, if not for themselves, at least for their children. You have to admit it's truly heartwarming to watch the young enjoy learning, especially knowledge to be proud of. (You) kids soak up this kind of stuff like a sponge, exploring far more advanced material far sooner than *we* ever did. And since there will always be children who will always become new enthusiasts, there will always be a need for another micro out there somewhere.

But the most fascinating sector of this surface must be the far right. I'd like to point out that the scenarios described previously have all been formulated from personal experience, as is the next. So many times I've talked with friends and acquaintances who are firmly convinced that the wave is still peaking, that micro proliferation is still on full charge! "Oh ya, that's really taking off right now, isn't it?", is a common response. Of course, *we* know the emphasis has faded. But what about this sector. Is this the untapped market? Untapped or not, it's out there – the proof is in the pudding!

Which brings about item two. Perhaps there is a market waiting to be tapped. What and who will unlock it remains to be seen, emphasis on "What". Because let's face it. . . new micros are coming, the Atari, the 128, the Amega, but they are not really new. Combinations of features and unbelievable prices don't make new technology. The question that sums it up best for me is, "Which company's stock value will go through the ceiling next?" The product belonging to that company is the one I'm waiting for. It will probably have a central processor, but it will leave the CPU as we know it in the dust. I call this "the next wave".

Then the far left will say, "let's get in". The inside left will have their "something new to be part of". The inside right may very well discard their indecision in favour of riding the wave. The average will finally have an alternative. The far right will complete the picture as Company X sweeps the continents with the latest "gotta have one" sensation, and *we'll* be in the middle, or even out in front!

In short, a new zone will emerge, one equally as big as *us*, and the cycle repeats. I don't profess to know when, where, how, or especially what. But I do know I wanna be there, I think we all do. Let's be ready.

There *is* nothing as constant as change

Karl J.H. Hildon, Managing Editor, I remain.

# 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 two versions of VERIFIZER on this page; one is for the PET, the other for the VIC or 64. Enter the applicable program and RUN it. If you get the message, "***** data 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 828 to enable the C64/VIC version (turn it off with SYS 831)
or SYS 634 to enable the PET version (turn it off with SYS 637)

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 it means we've editted that line at the last minute which changes the report code. However, this will only happen occasionally and 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 (eg. POKE 52381,0 instead of POKE 53281,0), but 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:** 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.

### Listing 1a: VERIFIZER for C64 and VIC–20

```
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
```

### Listing 1b: 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
```

# Bits and Pieces

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

## A Bunch of Disk Stuff. . .

### Disk Cleaner

**Peter Boisvert, Amherst MA**

*I clean my disk drive read/write head using a diskette–like insert containing a woven cloth disk impregnated with cleaning solution. To clean the head you must insert the diskette and close the door. Now the instructions say to "run the disk drive for 45–60 seconds" by sending any disk command to the drive. I used to use the initialize command. Unfortunately, the disk turns for only 4 seconds or so before it "knocks" the head and stops. To clean the disk properly requires repeating the disk command 10 to 12 times. That's an awful lot of knocking. Since too much knocking can precipitate head alignment problems, I was determined to find a better way. To my surprise the solution was very simple, provided you have a disk map of the ROM:*

```
10 rem* 1541 motor spin routine *
20 open 15,8,15
30 rem execute ml at $f97e to start motor
40 print#15,"m–e"chr$(126)chr$(249)
50 for i=1 to 6000:next: rem time delay
60 rem execute ml at $f9e8 to stop motor
70 print#15,"m–e"chr$(232)chr$(249)
80 close 15
```

*This short BASIC program executes two disk ROM routines directly, bypasssing the 1541 error checking protocol and avoiding the dreaded "knock". Location $F97E in disk ROM is the start of a routine which simply turns the drive motor on, nothing else. Similarly at location $F9E8 a routine exists which shuts off the drive motor. Thus all that is needed is a short program to execute the routines and a delay loop for the cleaning time. When the program is RUN the drive motor turns but the drive LED doesn't light. Ahh, the wonders of direct access programming! The motor will run for a minute and then stop, leaving a shiny disk in its wake. But, make sure the disk drive door is closed when the cleaning diskette is inserted, otherwise the head will not make good contact with the cleaning surface.*

Using Peter's technique, here's another 1541 motor spin program that will make it turn whenever the shift key is pressed. You can use SHIFT LOCK to keep the motor running if you wish. This one is handy when working on the drive.

```
10 rem* 1541 motor spin routine #2 *
20 print chr$(147)"hold SHIFT to spin drive motor"
30 print"press CTRL to quit program"
40 open 15,8,15
50 for i=0 to 1
60 s0=s1:s1=(peek(653)=1)
70 if s1 and not(s0) then
      print#15,"m–e"chr$(126)chr$(249): rem motor on
80 if not(s1) and s0 then
      print#15,"m–e"chr$(232)chr$(249): rem motor off
90 i=–(peek(653)=4): next: rem until ctrl pressed
100 close 15
```

### The 1541's amazing "*"

On the 1541, the special filename "*" can be used to load the most recently used file, or if no disk access has yet taken place, the first file on the disk. On other Commodore drives, "*" always loads the first file. If you want the 1541 to behave as the other drives, i.e. you want to load the first program on disk, just use the filename ":*" instead of "*", for example:

```
LOAD ":*",8
```

### World's Simplest Un–Scratch

The "*" filename on the 1541 will let you LOAD the last program SAVEd, even if it has been previously scratched! You probably won't believe it so try it for yourself:

SAVE the current program in memory: SAVE"0:TEMP",8
SCRATCH it from the disk: OPEN 1,8,15,"S0:TEMP"

You may check the directory at this point to make sure it has been scratched.

NEW the program in memory or even reset the C64 with SYS 64738 (don't turn it off and on, as this will also reset the 1541).

LOAD " * ",8 and your scratched program is back. Now you can safely save it again.

The above technique will not work if you've used any file since the scratched one, or if the drive has been reset. But it's great for those times when you realize you need a file right after you scratch it!

**C-64 Directory LOAD & RUN** **Bob Davis, Salina, Kansas**

*The 8032 series have the capability of using shifted RUN/STOP to load and run the first program on disk. . . but the 64 can go one better.*

*When you save a program, follow the program name with the following four characters:*

*1) A shifted space*
*2) Commodore D (The Commodore key and letter 'D' simultaneously)*
*3) Commodore U*
*4) Shifted '@'*

*This will force the disk directory to contain the file name in quotes, followed by ",8:" and all you do is display the directory, move the cursor to the appropriate line and press shifted RUN/STOP to load AND run your program.*

*While surely someone else has noticed this before, the trick is new to me, and I have not seen it published.*

**Jumbo Relative Files** **Elizabeth Deal, Malvern, PA**

*The B128 and the MPS–80 Drive* can *write large (500k) relative files without a "file too large" error. An old manual (circa 1982) has this incantation for the 8250, which just happens to work on the DOS 2.7 MPS drives:*

```
open 1,8,15
xx=0:print#3, "m–w" chr$(164)chr$(67)chr$(1)chr$(x)
close 1
```

*Reset, UJ or the above program with xx=255 turns the large-file feature off.*

*The CBM 8050 test/demo floppy has a program which expands relative files to an 8250 format. It works only on PET 4.0 computers; I don't have one. I find it mildly amusing that the 8050 test/demo wasn't fixed up to work on the B–machine.*

**APPENDing ML to BASIC**

A hybrid program – one using both machine language and BASIC – often consists of a single file on disk containing a BASIC program with machine code tacked onto the end. An easy way to create such a file is to simply SAVE the BASIC part, then send the object from your assembler to the same filename with the ",A" (append) filename extension. For example, using the PAL assembler:

```
100 open 1,8,12, "0:oldfile,p,a" :rem append to basic prg file
110 sys700  ;activate "PAL" assembler
120 .opt o1  ;direct object to append file
```

(The PAL example is redundant, since that assembler has hybrid capability, but you can use any assembler, or a BASIC loader program using DATA statements to generate the ML object.)

When Using this technique, the assembly origin will have to be set to the end of the BASIC program, which you can find by PEEKing the top-of-BASIC pointers ($2D,2E on VIC/64), and the new pointers will have to be set to the end of the ML object before you SAVE the BASIC (so that variables won't clobber the code). Also, remember that when using an assembler the first two bytes of the ML will be the start address, so you'll have to SYS two bytes past the start to execute the program.

**Another Use For ",A"**

The filename extension for append (,a) can help out when you're word processing. If you're creating a document and wish to maintain a table of contents, list of references, or any notes that come to mind, you can keep appending to a file by putting a ",s,a" or ",p,a" after the filename (depending on whether you're using SEQ or PRG files). Just set a "range" on the next note you wish to add to the file, and save the range with the above extension. Bits and pieces uses this technique with Superscript to keep a list of B&P authors in a separate file.

**Creating DEL Files** **David Stevenson, Pilot Mound, Man.**

*A "DEL" file may be created as follows:*

```
OPEN 2,8,2, "0:TEST,S,W"
OPEN 3,8,3, "0:TEST,S,W"
PRINT#2, "FIRST"
PRINT#3, "SECOND"
CLOSE 2: CLOSE 3
```

*The first file opened will become a DEL file. The DOS allows you to open more than one file with the same name as long as you haven't closed any and attempts to recover by giving a different file type designator. If you try this with more than two* *files all but the first two are lost. To make both files easily accessible just rename, changing the first one in the directory.*

*This happens with SEQ, PRG or USR files (or a combination) on my 1541. I haven't seen mention of this anywhere.*

Neither have we. It seems to work with the 8050 as well.

### Read Blocks Free Directly

This will let you directly read the number of blocks free on the current disk without any disk access (the disk must have been previously used in some way).

```
5 rem* read blocks free–1541
6 :
10 lo=250: hi=2: rem $02fa–$02fd
20 z$=chr$(0)
30 open 15,8,15
40 print#15,"m–r"chr$(lo)chr$(hi)chr$(4)
50 get#15,l0$,l1$,h0$,h1$
60 f0=asc(l0$+z$)+256*asc(h0$+z$)
70 print"blocks free:"f0
80 close 15
```

For the 8050 or 8250, make these changes (sorry, no 4040/2040 version):

```
10 lo=157: hi=67: rem $439d–$43a0
90 f1=asc(l1$+z$)+256*asc(h1$+z$)
100 print"blocks free – 0:"f0", 1:"f1
```

### 1541 Track Protect — John R. Menke, Mt. Vernon, IL

It's sometimes useful to be able to reserve certain tracks for later use, or prevent programs and files from being saved to a disk or certain tracks. Here's a short, quick 1541 utility which save–protects an entire disk or designated tracks. It works by writing zeros to the BAM (Block Availability Map), thereby misinforming the DOS that those tracks have already been used and are unavailable.

Conveniently, the BAM is restored and the save–protection removed simply by validating the disk.

```
ON  10 print"save–protect"
EN  20 print"(d) entire disk
IN  30 print"(t) a track
MO  40 geta$:ifa$=""then40
FH  50 if a$="d" then x=4:y=143: goto 100
MD  60 if a$<>"t" then 40
FE  70 input"track number";t
BB  80 if t<1 or t>35 then end
CM  90 x=t*4: y=x+3
CC  100 open 15,8,15
IK  110 open 5,8,5,"#"
PP  120 print#15,"u1:"5;0;18;0
MO  130 print#15,"b–p:"5;x
MN  140 for i=x to y
LJ  150 print#5,chr$(0);
EK  160 next
FD  170 print#15,"u2:"5;0;18;0
IM  180 print#15,"u;"
GC  190 close 5: close 15
JO  200 print"validate deprotects"
```

### Scratch & Save — Bob Hayes, Winnipeg, Man.

*Unlike SAVE with "@:", this program actually scratches your old file before saving the new one. I initially wrote it as an additional command to the TransBASIC language. Once the program is in memory, type this:*

SYS<start address>"filename"

*Notice there is no ",8" needed.*

Below are BASIC loader and PAL source listings of "Scratch & Save". The start address of these listings is $C000 (49152), but the program is fully relocatable. If you're using a dual drive, you'll have to remove lines 350 and 360 from the source code, and specify the drive number in the filename whenever you call "Scratch & Save".

```
PO  10 rem* data loader for "scratch & save" *
LI  20 cs=0
LF  30 for i=49152 to 49252:read a:poke i,a
DH  40 cs=cs+a:next i
GK  50 :
OC  60 if cs<>14558 then print"* data error ": end
MB  70 rem sys 49152"filename"
AF  80 end
IN  100 :
CB  1000 data  32, 158, 173,  32, 163, 182, 134, 251
BF  1010 data 132, 252,  72, 162,   0, 189,  90, 192
EC  1020 data  32, 210, 255, 232, 224,  11, 208, 245
MB  1030 data 169,   8,  32, 177, 255, 169, 111,  32
PG  1040 data 147, 255, 169,  83,  32, 168, 255, 169
AC  1050 data  58,  32, 168, 255, 104, 170, 160,   0
GH  1060 data 177, 251,  32, 168, 255,  32, 210, 255
MA  1070 data 200, 202, 208, 244, 132, 253,  32, 174
KF  1080 data 255, 165, 253, 166, 251, 164, 252,  32
OO  1090 data 189, 255, 169,   8, 168, 170,  32, 186
HL  1100 data 255, 169,  43, 166,  45, 164,  46,  76
GN  1110 data 216, 255,  83,  67,  82,  65,  84,  67
HK  1120 data  72,  73,  78,  71,  32
```

```
FD  100 sys700
HC  110 ; scratch and save
LP  120 ; bob hayes; winnipeg, manitoba
NI  130 ; routine help from brian munshaw's
AC  140 ; "new error wedge".
OP  150 .opt oo
MA  160 write  =    *
KB  170        jsr  $ad9e
KA  180        jsr  $b6a3
GD  190        stx  $fb
HE  200        sty  $fc
OE  210        pha
DC  220        ldx  #0
FF  230 mloop =     *
GD  240        lda  smsg,x
DJ  250        jsr  $ffd2
AO  260        inx
MP  270        cpx  #11
IP  280        bne  mloop
NB  290        lda  #8
FG  300        jsr  $ffb1    ;listen
DM  310        lda  #$6f
IO  320        jsr  $ff93    ;send secondary address
KJ  330        lda  #"s"
PD  340        jsr  $ffa8    ;ciout
DG  350        lda  #":"
DF  360        jsr  $ffa8    ;ciout
KP  370        pla
HE  380        tax
BN  390        ldy  #0
LA  400 sloop  =    *
IN  410        lda  ($fb),y
PI  420        jsr  $ffa8    ;ciout
HE  430        jsr  $ffd2
IJ  440        iny
JH  450        dex
CL  460        bne  sloop
IF  470        sty  $fd
JA  480        jsr  $ffae    ;unlsn
OM  490        lda  $fd
OC  500        ldx  $fb
PD  510        ldy  $fc
AB  515        jsr  $ffbd    ;setnam
DA  520        lda  #8
BO  530        tay
HO  540        tax
NO  550        jsr  $ffba    ;setlfs (open8,8,8)
BK  560        lda  #$2b
CF  570        ldx  $2d
DG  580        ldy  $2e
PJ  590        jmp  $ffd8    ;save $2b,2c to .x,.y
JJ  600 smsg .asc "scratching "
```

### C-64 POP

Sometimes you need to clean up the stack and re-start a program without killing variables, for example when you need to get back to the main menu from a deeply nested subroutine after an error condition occurs. The POP routine that works on the PET doesn't do the trick for the 64, but you can use this trick instead: just LOAD the program from within itself. That will cause an automatic re-run, cleaning the stack of subroutine return addresses and for..next loops, but leaving variables intact.

## Computer Stuff. . .

### C64/VIC20 PRINT AT Command — M. Van Bodegom, St. Albert, Alberta

*On many computers you can move the cursor to any spot on the screen with a simple command. For example, TAB(8,8) or PRINT AT(8,8); would allow you to print starting at row 8, column 8. Commodore doesn't have a BASIC command for this so most programmers PRINT down to the line and then use TAB(column). There is an easy way to get the cursor directly to any spot on the screen. The KERNEL has a routine that does just what we want. Simply use this line to set the cursor location:*

```
POKE 781,row: POKE 782,column:
SYS 65520: PRINT "message"
```

### Menu Select — Tim Buist, Grand Rapids, MI

There have been many menu selection programs, but this is one of the nicest to use, and it's fairly short! Just put the selections in the array 'A$()', the number of choices (up to 11) in 'N', then call this subroutine. It will display the options centred on the screen and highlight the first one. You can use the cursor up/down keys to highlight any option, and confirm the selection by pressing RETURN.

The subroutine returns with the chosen selection number in the variable 'I'. You can then branch the the appropriate section of your main program with ON I GOTO or ON I GOSUB. With the few additions given below, you can select using either the joystick or the keyboard.

```
100 rem* menu subroutine *
110 cd$=chr$(17): cu$=chr$(145)
115 hi$="r": off$="R"
116 rem use reverse-on and reverse-off for above,
117 rem  any two colours, or a combination.
120 aa=(25-n*2)/2: printchr$(147)
130 fori=1 to aa: print: next
140 fori=1ton: printtab(20-len(a$(i))/2);off$;a$(i):
      print: next
```

```
150 print chr$(19)
160 fori=1 to aa: print: next: i=1
170 printtab(20-len(a$(i))/2);hi$;a$(i)
175 get a$
180 if a$<>cd$ and a$<>cu$ and a$<>chr$(13) then 175
190 if a$=chr$(13) then return
200 printcu$;tab(20-len(a$(i))/2);off$;a$(i)
210 if a$=cd$ then print: i=i+1: if i>n then 150
220 if a$=cu$ then print cu$cu$cu$;: i=i-1: ifi<1then150
230 goto170
```

Notes:

1) Line 115 is set up to highlight the selected option with reverse field. If you wish, use colours for 'HI$' and 'OFF$', or colours combined with reverse on and reverse off (see comments in program).
2) To allow use of the joystick as well as the keyboard (up/down and fire to select), add the following lines:

```
176 j=peek(56320): rem 56321 for joystick port #1
177 if j=111 then a$=chr$(13)
178 if j=125 then a$=cd$
179 if j=126 then a$=cu$
```

## LIST Freeze

**Yijun Ding, Pittsburgh, PA**

Here's a real convenience utility. It lets you temporarily halt a program listing in progress to examine a section of code. Saves having to BREAK and re-list all the time! Once activated, this 21-byte machine language demon will live unobtrusively in your C-64 until you hold the SHIFT, CTRL, or Commodore key during a LIST to "freeze" the action. Just RUN the program below to set it up.

```
10 rem* data loader for "list freeze" *
20 cs=0
30 for i=49152 to 49172:read a:poke i,a
40 cs=cs+a:next i
50 :
60 if cs<>2031 then print"!data error!": end
65 sys 49152
70 print"qlist freeze activated.
80 print"qpress ctrl, shift or commodore keys
    to halt program listings.
90 end
100 :
1000 data 169,  11, 141,   6,   3, 169, 192, 141
1010 data   7,   3,  96,   8, 174, 141,   2, 208
1020 data 251,  40,  76,  26, 167
```

## A Couple of Plus/4 Goodies

Here are two pattern drawing programs that we borrowed from other magazines and adapted to the plus/4.

The first one, **Waving Spokes**, was originally designed to run on a Radio Shack plotter. You'll understand its title when you run it a few times. You can get vastly different patterns by supplying different parameters on start-up. Some recommendations: 20,6,20; 50,4,10; 30,6,60; 40,20,10; 20,4,100

After a pattern is complete, you can press F6 (RUN) to generate a new one.

```
1 rem" waving spokes - plus/4
2 rem" adapted from Bill and Lee Harding's
3 rem" program in Computek Magazine
4 :
10 graphic 0,1
20 input"no. of spokes, no. of waves, amplitude
    of waves";spok,waves,amp
30 graphic 1,1
35 p=360/spok
40 for angle=0 to 360-p step p
50 locate 160,100
60 for i=0 to 100 step 5
70 d=amp*sin(i*waves*.01745)
80 x=i*cos((angle+d)*.01745)
90 y=i*sin((angle+d)*.01745)
100 drawto 160+x,100+y
110 next i,angle
```

This next dazzler – **Kaleidoscope** – was originally written for an Atari machine. It's uncomplicated and easy to modify, but produces a constantly changing intricate pattern -- certainly worth a try.

```
1 rem" kaleidoscope - plus/4
2 rem" Adapted from kaleidoscope by
3 rem" Rafael Soriano
4 rem" in April '85 Atari Explorer
5 :
50 xm=159:ym=199:mc=1
60 graphic 3,1:color 0,1:color4,1:
    color1,8:color2,2:color3,4
65 do
70 for b=1 to xm
80 mc=mc+1:if mc>3 then mc=1
90 draw mc,b,c to xm-b,c
100 draw mc,b,c to xm-b,ym-c
110 draw mc,b,ym-c to xm-b,ym-c
120 draw mc,b,ym-c to xm-b,c
130 c=c+6:ifc>ymthenc=0
140 next b:color 3,4,i
150 i=(i+1)and7
160 loop
```

## BASIC Programming Tip – Simulated IF..THEN..ELSE

Here is a way you can put a statement on the same line as an IF. . .GOTO and have it execute if the branch *isn't* taken:

```
ON -(condition) GOTO 1000: statement(s)
```

This is equivalent to

```
IF (condition) THEN 1000: ELSE statement(s)
```

Since the C-64 and PET don't have an ELSE, the above trick can come in handy.

See why it works? By negating the condition, we get ON 1 or ON 0, which jumps to the given line if the condition is true, or "falls through" to the next statement if not. A bit tricky, but easier to follow than a rat's nest of GOTOs.

## ML Binary/ASCII Conversion Routines

**Tim Buist, Grand Rapids, MI**

This first routine is easy to use: just place the binary number you wish to convert after the SYS, for example:

```
SYS 49152, 110010
```

The 16-bit result will be in RESULT and RESULT + 1, which are 828 and 829 in the listing below.

```
100  sys700;pal 64 assembler
101  ;this program converts an ascii
102  ;binary number to actual binary
103  ;form and stores it in "RESULT"
104  ;it works on anything up to 16 bits
105  ;
110  .opt oo
120  result    =    828
130            lda  #0         ;clear it first!
140            sta  result     ;lsb
150            sta  result+1   ;msb
160  loop      =    *
170            jsr  $0073      ;chrget
180            cmp  #"0"
190            beq  zero
200            cmp  #"1"
210            beq  one
220            rts             ;return if not 0 or 1
230  zero      =    *
240            clc
250  one       =    *
260            rol  result     ;put in carry bit
270            rol  result+1
280            jmp  loop       ;get more digits
```

*While looking like it does nothing, it actually rotates a bit into RESULT. Since a CMP. . .BEQ will sett the carry bit, at ONE the carry bit will be ROLed into RESULT. If the CMP #'0' succeeds, the carry bit is cleared and a zero inserted into RESULT. These Sure are fun to write!*

*Here's another simple but fun subroutine that converts an 8-bit binary number to ASCII binary and prints it. While this is again a not-so-complicated-that-I-couldn't-think-of-it subroutine, it might spark someone just getting started in M.L.*

```
100  sys700;pal 64
101  ;this program converts a byte
102  ;to its ascii binary equivalent
103  ;and prints it.
105  ;
110  .opt oo
120  number    =    828        ;result will go here
130            ldx  #7         ;8 bits
140  loop      =    *
150            lda  #"0"
160            asl  number     ;get a bit from number
170            adc  #0         ;add in carry
180            jsr  $ffd2      ;print it
190            dex             ;next bit
200            bpl  loop       ;all 8 bits done"?
210            rts
```

## Lett'er Fly!

Try this:

```
10 s1$=chr$(19)+chr$(17)+chr$(157)
   : s2$=chr$(19)+chr$(29)+chr$(20)
20 get a$
30 print s1$a$s2$: goto 20
```

Press a few letter keys and watch. We know, neat but totally useless, right? Well, modify line 20 like this:

```
20 get a$: if a$ = "" then 20
```

Now try it. You might have a use for an input routine like that in one of your programs.

# Letters

**Just Love Those Transactor Disks:** As you remarked in your comment at the end of David W. Tamkin's letter published in the July 1985 issue, whether to get the programs from Transactor already on disk saved or to type them in for yourself from the listings given in the magazine is reader's choice.

What Mr. Tamkin obviously does not realize is that there are many reasons why a reader either may not be able to type them in – or even why it may be impossible for him to type them in correctly!

Victims of dyslexia are far more common than perhaps people realize. The commonest form of this reading problem is the reversal of the ORDER in which the reader sees a small set of letters or digits. He does NOT see the mirror images of these characters – it is only their order that gets reversed. The use of checksums does not help such people, for changes in the order in which characters are typed does NOT cause a checksum error.

Also there are many types of disabilities affecting the use of their hands. For many, typing in long programs is exhausting and so very difficult because exhaustion increases the already high error rate due to the disability the typist has.

There can be other reasons why it is undesirable or impossible for a reader to find the time to put in the hours required to type in these programs and then debugging them.

Being a dyslexia victim, I have had to ask authors of programs what they would charge me to copy their programs upon a disk which I would provide – just to get a very few of the programs which were not sold be dealers or software firms that I needed to use. Having had one arm totally paralyzed – though I was one of the fortunate few who in time recovered full use of that arm – I know how impossible it would have been for me to type in any long program while that paralysis was wrecking even hunt–and–peck typing for me.

The publication or non–publication of disks of programs contained in an article is irrelevant to the level of expertise assumed for a magazine's readership or for its quality. Making such disks available does, however, show concern on the part of the editors and publishers for the problems some of their readers may have with respect to using the programs listed in their magazines.

I wish to congratulate and thank you, the publishers and editors of Transactor, for making the Transactor Disks available to your readers if they wish to order them.

Mrs. Marge Paulie, Eugene, Oregon

*It may interest you to know that Mr. Tamkin called us shortly after his letter but before that issue hit the newstands. After apologizing for the letter (which, by the way, was unnecessary) he proceeded to order disks. To be quite honest, though, it hadn't occurred to us that our disks would benefit the disabled moreover others. Thank you for pointing that out to us. Making Transactors is a lot of work and a lot of fun, but letters like yours help tip the balance that much more towards the latter. Thank you again.*

**Ad-vice:** Hey, Transactor, you're missing the boat! You boast a print run every issue of 64,000 copies. Compute! boasts about 600,000. You print about 75 pages every issue of terrific information for the Commodore enthusiast, Compute! has now dropped to 96 pages, with a 50% advertising content. That means that a maximum of 48 pages contain actual usable info. To further water down the content, these 48 pages are divided up between Commodore, Atari, Apple, IBM, and TI. When tallied up, a very small portion of each Compute! would be of use to most Commodore users. Now, Commodore users dominate the home computer arena. There are millions of them out there. It stands to reason that many of Compute!s readers are Commodore users, with many users buying both Compute! and Compute!s Gazette. Chances are that most of their readers have never even heard of The Transactor.

There seems to be two ways in which to increase your sales figures. The first is to update your marketing strategy to include advertisements in as many Commodore related magazines as possible. The second, and possibly the most effective, is to ask your readers to spread the word of The Transactor as far and wide as possible. It does not take a lot of grey matter to realize that the only way to make a virtual advertising free magazine pay off is to increase the subscriber base as much as possible. Magazine rack sales may sell a lot of magazines for you, but they also force you to reduce your prices to your distributors, give terms on payment, and allow a return policy for unsold magazines. Subscription sales, due to the fact that payment is immediate, in advance, and in full, is where the profits are. The only major expense to you is mailing out the magazine every issue. Boost your subscriber base, and you will be on easy street.

One more bit of advice before I sign off. Advertising. Why not bring it back again. Ads are only offensive when they are splattered everywhere, as most magazines do. I like to read ads, but not while I am reading an article. Your concept of placing the ads in the back, and once in a while at the very front, is terrific. It's not offensive, and encourages me to read them at my own leisure. Your sales figures are up since you dropped your ads, so why not re–introduce them once again. An increase in your ad content could possibly be the key to a greater Transactor future.

John Brunner, Chicago, Illinois

*You either have ESP or you've been eavesdropping on our headquarters via long distance. We have been trying to dream up ways to increase our subscriber base since day one. To date we've been fairly successful, with a base right now of about 10,000.*

*At present there are Transactor ads in some of the Commodore related mags but some will not accept ads from what they deem "competitors". Also, we have been getting a lot of support from quite a few of the users groups everywhere, with mentions in their newsletters, and messages mysteriously appearing on BBS's all over. This hasn't hurt our sales one bit. But we would always appreciate anyone passing the word. Increase our sales figures and you will earn our affection forever.*

*About bringing advertising back. We have debated it, and have decided to bring them back, in limited quantity. We still want to try to keep the magazine 90% ad free. We will offer a total of seven, full page ads to run, laid out in the magazine as would a second cover on the inside. Hopefully, our readers will enjoy the ads as much as you seem to.*

*Thanks for the terrific advice, and we hope that if any more helpful thoughts pop up, you will drop us a letter.*

**A Few Notes On DOS:** Congratulations on "Learning The Language Of DOS" in Vol. 5, No. 5, which I found interesting and useful. With your tips I quickly converted my custom, homebrew disassembler to work with the 1541 RAM/ROM. I look forward to more 1541 memory maps, but encourage that the tabular size be made larger than on page 51 of the above issue to spare my eyesight.

It isn't quite sufficient to say that "B–R", "B–W", "B–A", and "B–F" are tainted, and I hope that you will mention why; I have had no problems with "B–W". But, indeed, "B–R" doesn't seem to pay attention to the Buffer–Pointer and simply reads the first few bytes of a block then stops. In agreement with your experience, I have noted no other problems with "B–P". I haven't had enough experience with "B–A" or "B–F" to make a judgement yet. "UI" and "UJ" seem to work OK, but I haven't tried the alternate syntax for the other user commands. It's easy enough to accept your advice to use the standard syntax here, if you will be a bit more specific about the reasons.

Let me mention a caution with relative files. If a relative file is left open, inadvertently such as during program development, the 1541 DOS crashes! I lost a good disk that way. Initialization (@I) of the drive doesn't fix things and subsequent disk operations will damage other disk files. In this case, a save with replacement leaves the disk directory looking like scrambled eggs; you can't even format a new disk. The cure is a reset with "UJ" or by turning the disk drive off and on.

I have yet figured out how to write to a relative file in emulation of the "U2" command. So far as I can determine, an entire relative file record must be read into computer memory, updated there, then all fields of the record must be rewritten to disk. It would be faster and generally more useful if there were a way to emulate the "U1" then "U2" sequence, normally used with random files, with relative files. I suspect that there may be an easy way to do so, but I haven't yet stumbled across it after a lot of syntax and command experiments. The only thing I've come up with is to access and interpret a side sector to get the track and sector of a record number; then random file commands are handy enough if the record length happens to be exactly 254 or 127 bytes.

I note that when a relative file record is accessed, apparently the 1541 DOS reads two disk sectors into disk RAM. This observation might be useful now that you have kindly published the addresses of the RAM buffers! I haven't yet checked to see which two buffers are involved. Thus, relative file records of 127, 254, 381, or 508 bytes in effective length can probably be efficiently constructed with the 1541 relative file system.

I note that 'Single Disk Copy Program' by Rick Illes, on pages 13–14 of Vol. 5, No. 5, doesn't work on my Commodore 64. There's a typo in program line 130 : PEEK(46) should be PEEK(56). I can't say whether anything else is wrong because I reworked thing extensively from this point on.

John Menke, Mt. Vernon, Illinois

*To begin, thanks for the voice of approval regarding my article. It took a while to write, but from the sounds of things since then, many people have enjoyed it. To be a bit more specific, "B–W" has been blamed in the past for clogging up the error channel in use. The syntax of a reset, "UJ", "U:", or "U;", seems to vary depending on the ROM revision you have with your 1541. You will know what doesn't work for you when your drive hangs up through its use. "B–A" and "B–F" again are dependent on ROM revision. Older ROM's seem to have the problem of them not working in general. It seems that Commodore has always had some difficulties with these two. One piece of advice, lifted right out of Commodore Magazine of February 1982, is to convert all numerals into strings and concatenate them into the command string before issuing the command. Most people write their own Block–Allocate and Block–Free routines to synthesize the process in computer RAM. This technique is my favourite because you are always sure that it took.*

*Sorry for the type size, but it was the only way to fit it on the page. That map was really there to give you a taste for our 'Complete Commodore Inner Space Anthology', to incite you to dash out and buy it. Inside this oddly named book we have placed the ROM/RAM maps plus definitions for the 1541, 4040, and 8050 drives. Interested yet?*

*Thanks for the advice regarding relative files, and their crashes thereafter. I really didn't know that this problem existed. Let's hope that some brilliant disk doctor out there takes your hints and comes up with a synthetic relative file maker just for you. Might be a neat application.*

**A Bit More DOS Advice:** I'd like to comment on some of the statements in "Learning The Language Of DOS", by Richard Evers in the March '85 issue, and make some corrections to the '1541 User's Manual'.

My first argument is with the statement on page 48 of the article that claims that Block Commands (Block–Allocate, Free, Read, and Write) are "terminally ill". Perhaps there were some problems with the original 1540 ROM's. However, all of the commands do work flawlessly. I have written, used, and distributed several programs that rely on these commands, and they have never made any mistakes. I feel a major reason for the confusion with these commands is the User's Manual. On page 29, the format for Block–Allocate is shown: PRINT#file#,"B–A:"drive,track,block. Only the first comma is correct, the rest should be semi–colons. That is, PRINT#file#,"B–A:"drive;track;block. All other Block commands are listed incorrectly as well. The correct usage is shown on page 41 of the manual, and in the article. Also, it is not necessary to close the command channel after using any command, IF you use them correctly. Overall I found this timely article to be both informative and useful. I appreciate the technical aspects of your magazine, and I hope it remains that way.

The manual included with the 1541 drive has enough bugs to keep the experts guessing until the technology becomes techno–obsolete. Few of the tutorial program work as written. An almost ridiculous error on page 8 is a good example of the writer's carelessness. It reads, "never remove the diskette when the green drive light is on". Of course, they meant red, didn't they? On page 4, Commodores manual claims the 1541 is write compatible with both the 4040 and 2031 disk drives. Perhaps it is in theory, but it's never worked for me or my friends. There have been rumours that Commodore has written a new manual for the 1541. If they have. I strongly suggest you try to get one. It might clear up a lot of Head Aches.

One final rumour about the 1541 is a fault in the save–with–replace command (save"@0:filename"). I, too, blamed it for destroying my programs and data. But I discovered the real culprit was an occasional disk swap, forgetting to 0 after the @, or, worst of all, absent mindedly typing save"s0;filename". Since I started double checking my typing and initializing the drive each time a disk swap was made, I have had no problems. Remember, too, if you don't give each disk you format a unique ID, just changing the disks can be fatal.

I recently had a chance to use a new 1541 drive. They have a "right–angle" door latch, no over–heating problems, no head–banging (suggesting a new ROM).

Still, if you're in the market for a new Commodore compatible drive, you might consider the Commodore 128's 1571 multi–mode disk drive. It behaves like a 1541 in the 64 mode, and can be directly connected to the serial port. In the 128 mode (for use with the C128 only), it becomes a dual–sided (340K Byte) drive capable of speeds of 12000 baud! That's more than 46 times faster than a normal 1541. It's also able to read CP/M disks when used with the C128. For more information, see Commodore Magazine, April 1985.

Tom Johnson, Jefferson, Missouri

*Commodore documentation always seems to have bugs in it, regardless of who it's written by (ie. Commodore or otherwise) and I suppose no manufacturer is 100% immune to this problem. In defence of my statements regarding the terminally ill Block Commands, I still feel that some revisions of the 1541's ROM's are still a little shakey. Also, look back at my article once again. Closing the command channel after access was only specified with Block–Allocate and Block–Free. Other than that, your letter is terrific. Oh, by the way. Hope you've been catching our current debate regarding the save with replace bug. Charles Whittern was able to reproduce it, but not isolate its cause.*

**18–0 Screwup Fixed:** Having read your article on "Learning The Language Of DOS" in Vol.5, Iss.5 of The Transactor, I am now apparently one of those dangerous people. (You know what is said about someone with a little knowledge?) Without dragging out a story, here's my situation briefly:

A friend of mine has a program called "18–0 Screwup". Believe me, it works JUST FINE! He inadvertently ran it while he had a disk in his drive which he didn't want screwed up. It appears that only the second and perhaps third byte of track 18, sector 0, has been changed. He asked me for help. So, armed with your article and the 1541 drive manual, I set to work. Enclosed you'll find the short program I've been trying. I have narrowed the problem down to around line 150. No matter what I've tried (closing unnecessary channels, using a different channel from the one used for the "B–R" command, and replacing the "B–W" command with the "UB" and "U2" command), I still get the error 70, NO CHANNEL.

Can you help? The disk in question is not a critical one, as there are back–ups on file, but, now it has become a riddle to me. Any input you can give will be welcome.

```
50 open 15,8,15, " i "
60 open 5,8,5, " #0 "
70 print#15, " b-r: " 5;0;18;0
75 close 5
80 print#15, " m-r " chr$(1)chr$(3)
82 get#15,a$
83 print asc(a$ + chr$(0))
90 open 8,8,15
100 print#8, " m-w " chr$(1)chr$(3)chr$(1)chr$(1)
110 print#15, " m-r " chr$(01)chr$(03)
120 print#15, " m-r " chr$(1)chr$(3)
130 get#15,a$
140 print asc(a$ + chr$(0))
145 open 5,8,5
150 print#15, " b-w: " 5;0;18;0
160 close8: close5: close15
```

Dennis McKee, Ottawa, Ohio

*The problem with a No Channel Error is one that I am familiar with. It caused me great pains a long time ago when first working directly with Commodore DOS. It took quite a bit of experimentation, just as you have done, before the cure was found. The cure, do not initialize the drive when OPENing the 15th channel. The bug is that once the drive starts initializing, it tends to ignore a few commands coming over the bus, namely the OPEN statement. In your example, as in my original one, OPENing the direct access buffer through channel 5 was ignored, therefore a No Channel error would be generated thereafter through reference to channel 5. If you leave out the Initialization, your program should work.*

*If you care to key in the program listed below, you might find it worth your effort. It's a take off of your program, with a few mods. It reads track 18, sector 0 into RAM buffer #0, $0300. Next, it displays the first 3 bytes held in the buffer. These bytes will normally be 18, 1, and 65. The 18 and 1 point to track 18, sector 1, the first directory block. The 65, ascii 'a', represents the DOS format, 1541/2031/4040. If you were to change this 65 to any other value, you might find a bit of fun waiting. You would not be able to write to the diskette any more, nor scratch files, quick new the diskette, or even Back-Up the diskette if using a 4040 drive. This trick has been mentioned before in an article/ program I wrote a while ago called 'Drive Protect'.*

*To get back on track, following the display of the current contents at that location, you are given a prompt to update RAM (y/n). Any other response but 'y' at this point will abort the program. Once the RAM has been updated, the new data held at $0300–$0302 will be displayed, just for your peace of mind. Another prompt will then materialize, asking if you really want to write the block back to the diskette. As before, anything but a 'y' will abort. Once the block has been correctly written to diskette, the files are all closed up, and the program ends. A nice ending to a bad experience.*

```
100 rem save "0:18-0 un-screw",8
105 z$ = chr$(0)
110 open 15,8,15: open 5,8,5,"#0"
115 print#15,"u1:"5;0;18;0: rem * read in track 18,
     sector 0
120 print#15,"m-r"chr$(0)chr$(3)chr$(3)
     : rem * peek about in ram
125 for x = 0 to 2: get#15,a$: print 300 + x;asc(a$ + z$)
     : next x
130 input "** update ram (y/n) ";sr$: if sr$<>"y" then 160
135 print#15,"m-w"chr$(0)chr$(3)chr$(3)chr$(18)
     chr$(1)chr$(65)
140 print#15,"m-r"chr$(0)chr$(3)chr$(3)
145 for x = 0 to 2: get#15,a$: print 300 + x;asc(a$ + z$)
     : next x
150 input "** write back block (y/n) ";wb$
     : if wb$<>"y" then 160
155 print#15,"u2:"5;0;18;0: rem * write back to track 18,
     sector 0
160 close5: close15: end
```

**Long Lost PAL:** Today I discovered, to my satisfaction, a super magazine dealing with the things I want to know. I can foresee a subscription to Transactor would be money put to wise use, and in the near future such a thing will happen.

In the meantime, please enlighten us new comers to your publication. You mention the PAL assembler by Brad Templeton. Where can we find this assembler and how much should we expect to pay?

This PAL sounds like a super good assembler, why haven't we heard about it in Washington.

Brad Moore, Seattle, Washington

*It's nice to know that we're appreciated. The PAL Assembler is possibly the nicest assembler that you will ever work with on the Commodore machines. The syntax is similar to that of the Commodore Assembler, but it has some pretty sharp additives. The reason why you haven't heard of it in Washington is possibly because no dealers out your way have either. Try the address below for a copy, worth $69.95 Canadian.*

*Pro-Line Software*
*755 The Queensway East, Unit 8*
*Mississauga, Ontario*
*L4Y 4C5 (416) 273-6350*

**Chop, Goes The Executor:** I am writing regarding an article appearing in the July, 1985, issue of The Transactor, called DOS FILE EXECUTOR by Chris Johnsen. First, I would like to say how pleased I was to see you tackle this hidden feature of the 1541 drive. I would very much like to see more articles of this kind!

There are a couple of problems with Mr. Johnsen's program as it was published. The most important is that it will not create proper DOS EXEC FILES if the program is longer that 250 bytes! What is missing is an update of the LOW/HIGH address of $00/03 instead of $00/03, $FA/03, $F4,04, etc.. What happens is that each block is loaded into successive buffers and then overlaid onto buffer 0 ($0300) leading to massive confusion.

A tip that your readers might find useful when working with DOS EXEC FILES is to place an RTS ($60) in front of the first byte in your M/L routine before creating a DOS EXEC FILE of it (Or modify Mr. Johnsen's program to do it for you!).

This will allow you to 'park' your main routine in the drive and have control of it returned to you without its being executed. This is useful because you may first need to memory-write (M-W) values to the drive and also want to memory-execute (M-E) at a different location.

Bill MacMillan, Prince George, British Columbia

# TransBASIC Installment #5

Nick Sullivan
Scarborough, Ont.

TransBASIC has been generating a lot of mail, lately, and I would like to thank all of you who have written in with your problems, questions, suggestions and -- yes, new TransBASIC modules, some of which appear in this issue. Before we get to those, though, let's take a look at the rest of the mail.

### Assembler Compatibility

Several readers have had success in adapting TransBASIC to assemblers other than PAL. One common requirement is to change PAL's non-standard .asc pseudo-op, with double quotes, to .byte with single quotes.

Not all assemblers parse expressions in the same way PAL does. For instance, given the instruction:

```
lda #>label-1
```

. . . the effect in PAL is to load the accumulator with the high byte of the address ('label-1'). At least one assembler, the Commodore 64 Macro Assembler Development System, evidently takes a different approach, by first taking the high byte of 'label', and then subtracting 1. Presumably the answer is caution and parentheses:

```
lda #>(label-1)
```

I would appreciate it if readers would let me know of other problems along this line.

Back in the first TransBASIC column, I said that "unless you have access to a copy of PAL, or some other assembler that parasitizes the BASIC source editor, TransBASIC is not for you". After receiving a letter asking for an elucidation of that remark, I realized it was a bit too sweeping. The point was that the ADD command will merge TransBASIC modules only if they are stored in the form of BASIC program text -- assemblers with their own editors won't work. On the other hand, if the particular package offers some means of merging files by line numbers, the ADD command isn't necessary, and maybe TransBASIC is for you after all.

### Bug Reports

Numerous letters make mention of three problems. 1) The shifted left parenthesis was missing from the keyword line (602) in the CHECK & AWAIT module that appeared in instalment 2. The line should have read:

```
602 .asc "check": .byte $a8: .asc "await": .byte $a8
```

Originally, this line was written with graphics characters embedded in the .asc string, and no .byte commands, but this is difficult to reproduce in a typeset program listing. 2) The CURSOR POSITION module, which was supposed to have appeared in the second instalment, didn't actually make it until the third. 3) Early copies of the Transactor disk with the programs of instalment number one, had a problem with the TransBASIC loader program. In the incorrect copies, line 130 of this program reads:

```
130 a=1: load "tb/add.m",8,1
```

The correct version is:

```
130 a=1: load "tb/add.obj",8,1
```

Now for a trickier bug. David Stevenson of Pilot Mound, Manitoba, correctly points out that the indirect jumps in the TransBASIC kernel (tvec, lvec, evec and fvec) could potentially lie across a page boundary, depending on the size of the keyword table. Owing to a bug in the 6502/6510 microprocessors, this condition would cause a crash. The solution is to make sure that the vectors fall on even-number memory locations, or that they do not lie across a page boundary.

Taking the latter approach, Mr. Stevenson suggests putting the vectors before the keyword list instead of after. This would mean changing the line numbers around, but could be done fairly easily. Or, you could add the following line to the kernel:

```
2129 .if >(*&255)+7: *=*+(*&1)
```

This rather cryptic line will pad your object code by one byte if the vector table that follows would otherwise lie on an odd byte and across a page boundary. The number 7 represents the number of bytes in the table minus one -- by choosing the appropriate value you could use this line any time a vector or a table of vectors occurs in a program you are writing. Will it work with assemblers other than PAL? I don't know.

### New Modules

Six of the seven modules published this issue were contributed by readers, and there are more to come. I have edited all of them, sometimes heavily, to mesh more closely with TransBASIC; I hope I have not introduced any bugs.

The LABELS module comes from Jerry Gillaspie of North Hollywood, California. Mr. Gillaspie writes: "I have always felt that the biggest problem with BASIC was the need to GOTO and GOSUB to a line number. The line numbers have no significance relative to the function being performed." His new commands, L., LGOTO and LGOSUB get around this problem nicely. I added SGOTO and SGOSUB to the module for even greater flexibility -- and introduced a problem. This is dealt with in another small module, TOKEN & VAR.

Charles Kluepfel of Bloomfield New Jersey, has contributed two modules. One, ARCFUNCTIONS, provides two trigonometric functions missing in regular BASIC. The other, INSTRING, duplicates the INSTR( function found in many BASICs, but with an extension that makes use of the Boolean operators.

Mr. Kluepfel asks an interesting question about compatibility between TransBASIC dialects: "If I write a program on a (dialect) having commands A, B, and C, utilizing the B and C commands, then later try running on a version that has B, C and D, the B and C commands will have different tokens, and the thing won't work."

This is entirely true. The whole point of TransBASIC is that keywords are dynamically, not statically, assigned to tokens. Thus, in different dialects, the same keyword may have a different token. There are two answers to this difficulty. One is to make a new dialect for every new program you write, to label it, and to stick with it. The other is to search and replace tokens with a programming utility. That can get you out of a jam, but it's a lot more awkward.

Mr. Kluepfel adds: "As for other commands and functions I would like to see, these include PRINT USING, SWAP (interchange two variables), UNDIM (to delete one or more arrays) from memory so it can be reDIMmed), a new RND that allows specification of the range of random numbers desired or a repetition of the previous random number given, a RESTORE to a line number, a LINPUT, and a computed GOTO."

Anyone interested? We already have one version of a SWAP command awaiting publication, and a version of the RND function similar to the one Mr. Kluepfel suggest, but without the repetition feature. An extended INPUT has also been written, that does not produce the question-mark prompt, and can be terminated only by a carriage return. Of course, the INPUT statement has always provided lots of room for innovation, and there are plenty of other possibilities. The UNDIM will require a memory move utility, one of which will be introduced in the next column, so it might be best to hold off on that for now.

Another 'instring' function comes from Michael Phillips of Camden, Tennessee. This one also features an interesting extension: the ability to specify a point in the first string at which the search for the second string is to begin. In order to distinguish it from Charles Kluepfel's contribution, I renamed this one PLACE(, as in Simons' BASIC.

Shaun Erickson of Jamestown, North Dakota, has sent in the PRINTAT module, which is like an extended version of the CURSOR command.

And Frank Vanzeist, of St. Mary's, Ontario, has contributed his extensive SOUND THINGS module, with its 28 statements and 4 functions, which should make poking the SID chip a thing of the past.

Thanks to all the above contributors, and to those whose work has been received, but not yet published. Next issue, I hope to have some disk commands by Darren Spruyt, whose work has often appeared in this magazine in the past; a very fast merge routine that you can use instead of ADD; and much more.

## New Commands

This part of the TransBASIC column is devoted to describing the new commands that will be added each issue. The descriptions follow a standard format:

The first line gives the command keyword, the type (statement or function), and a three digit serial number.

The second line gives the line range allotted to the execution routine for the command.

The third line gives the module in which the command is included.

The fourth line (and the following lines, if necessary) demonstrate the command syntax.

The remaining lines describe the command.

**L.** (Type: Statement Cat #: 073)
Line Range: Routine in ROM
Module: LABELS
Example: L.GETLOOP: GET U$
A line is labelled for reference by the LGOTO, LGOSUB, SGOTO and SGOSUB statements. The L. command must be the first on its program line if the label is to be recognized.

**LGOTO** (Type: Statement Cat #: 074)
Line Range: 5924-6100
Module: LABELS
Example: IF A$<>CHR$(13) THEN LGOTO GETLOOP
The program is searched for a line bearing the specified label. If found, execution continues from that line, otherwise an Undefined Statement error results.

**LGOSUB** (Type: Statement Cat #: 075)
Line Range: 5870-6100
Module: LABELS
Example: LGOSUB BLUEBIRD
The program is searched for a subroutine labelled as specified.

**SGOTO** (Type: Statement Cat #: 076)
Line Range: 5920–6130
Module: LABELS
Example: U$ = " BLUEBIRD " : SGOTO U$
The program is searched for a line bearing the label specified by the string expression. If found, execution continues from that line. Otherwise, the program is searched for a line with the label DFAULT, and if found, execution continues from there. Otherwise, an Undefined Statement error results.

**SGOSUB** (Type: Statement Cat #: 077)
Line Range: 5866–6130
Module: LABELS
Example: INPUT L$: SGOSUB L$
The program is searched for a subroutine bearing the label specified by the string expression. If found, the subroutine is executed. Otherwise, the program is searched for a subroutine with the label DFAULT and, if found, the subroutine is executed. Otherwise, an Undefined Statement error results.

**TOKEN$(** (Type: Function Cat #: 078)
Line Range: 6132–6196
Module: TOKEN & VAR
Example: SGOTO TOKEN$( " POKER " )
A string is returned which is the tokenized version of the argument string. One use is illustrated in the example. The label specified by the L. labelling command (073) is tokenized by the BASIC and TransBASIC tokenizing routines, whereas the argument string of the SGOTO and SGOSUB commands is not tokenized. This would result in the label not being recognized if it contains one or more BASIC and/or TransBASIC keywords (as with " POKER " ). By tokenizing the string with this function before the search, the match can be made successfully.

**VAR(** (Type: Function Cat #: 079)
Line Range: 6198–6208
Module: TOKEN & VAR
Example: PRINT VAR(U$)
An address is returned corresponding to the address of the data in the named variable -- the third byte in the variable's entry in the table above BASIC program text space. In the case of numeric variables, the address is that of the actual data; in the case of string variables, the address is that of the string descriptor.

**INSTR(** (Type: Function Cat #: 080)
Line Range: 6210–6396
Module: INSTRING
Example: A = INSTR(U$,V$)
Example: B = INSTR( " INSANE " , " SANE " ,AND)
Example: IF INSTR(W$, " JKQXZ " ,OR) THEN PRINT " GOOD SCRABBLE WORD "
Example: IF INSTR(M$, " 01 " ,NOT) THEN PRINT " NOT BINARY " )
String 1 is scanned for an occurrence of String 2. If one is found, the starting position of String 2 in String 1 is returned, counting from 1. An unsuccessful search returns 0. The search can be modified by using a Boolean operator as the third argument in the function. AND is the default, and operates as described above; therefore example two returns the value 3. OR returns the position of the first character in String 1 that matches any character in String 2. NOT returns the position of the first character in String 1 that does not match any character in String 2.

**PLACE(** (Type: Function Cat #: 081)
Line Range: 6398–6546
Module(s): PLACE
Example: Q = PLACE( " CLOVERLEAF " , " LOVER " )
Example: R = PLACE(5, " RAT-A-TAT-TAT " , " AT " )String 1 is scanned for an occurrence of String 2. If one is found, the starting position of String 2 in String 1 is returned, counting from 1. An unsuccessful search returns 0. The position in String 1 at which the search is to commence can be specified with an optional first parameter as in the second example, which returns a value of 8.

**ASN(** (Type: Function Cat #: 082)
Line Range: 6548–6702
Module: ARCFUNCTIONS
Example: U = ASN(1/2)
The arcsine (inverse sine) of the argument is returned. Arguments less than -1 or greater than +1 are illegal quantities, except that the function is forgiving of quantities exceeding 1 in absolute value, but very close to it, counting them as equal to 1 to allow for accumulated errors in trigonometric computation.

**ACS(** (Type: Function Cat #: 083)
Line Range: 6670–6702
Module: ARCFUNCTIONS
Example: U = ACS(V/W)
The arccosine (inverse cosine) of the argument is returned. Arguments less than -1 or greater than +1 are illegal quantities, except that the function is forgiving of quantities exceeding 1 in absolute value, but very close to it, counting them as equal to 1 to allow for accumulated errors in trigonometric computation.

**PRINT@** (Type: Statement Cat #: 084)
Line Range: 6704–6744
Module: PRINTAT
Example: PRINT@ 15,5, " FLEAS IRK US "
Example: PRINT@ 5,12: INPUT C$
The cursor is moved to the specified column (first argument) and row (second argument), and the third argument, if any, is printed at that position. The third argument is passed directly to the BASIC print routine, and can be anything that is legal in a PRINT statement.

**CLESID** (Type: Statement Cat #: 085)
Line Range: 6908–6922
Module: SOUND THINGS
Example: CLESID
Clears the 25 write only registers of the SID chip, and the SID image maintained by the SOUND THINGS module.

**FREQ** (Type: Statement Cat #: 086)
Line Range: 6924–6932
Module: SOUND THINGS
Example: FREQ4,53000
The first argument, in this and other SOUND THINGS commands, specifies the voice(s) to which the command is to apply. The argument is a 3-bit value in which the state of each bit indicates whether the corresponding voice is included in the command. The number 4, in the example, indicates that in this instance the command applies only to the third SID voice. An argument of 5 would cause the command to affect both the first and the third voice; 7 would affect all three voices. The second argument is a frequency to be poked into the frequency registers for the indicated voice(s).

**PUWID** (Type: Statement Cat #: 087
Line Range: 6934–6948
Module: SOUND THINGS
Example: PUWID3,1000
Set the pulse width (second argument) of the voices specified in the first argument.

**FIFREQ** (Type: Statement Cat #: 088)
Line Range: 6950–6978
Module: SOUND THINGS
Example: FIFREQ FF+I
Set the filter cutoff frequency to the specified value.

**ADPUL** (Type: Statement Cat #: 089)
Line Range: 6980–7026
Module: SOUND THINGS
Example: ADPUL 2
Switch on the pulse width wave form in the specified voice(s), without affecting other bits in the wave form register except the noise bit, which is cleared.

**ADSAW** (Type: Statement Cat #: 090)
Line Range: 6984–7026
Module: SOUND THINGS
Example: ADSAW 6
Switch on the sawtooth wave form in the specified voice(s), without affecting other bits in the wave form registers except the noise bit, which is cleared.

**ADTRI** (Type: Statement Cat #: 091)
Line Range: 6988–7026
Modules: SOUND THINGS
Example: ADTRI 7

Switch on the triangle wave form in the specified voice(s), without affecting other bits in the wave form registers except the noise bit, which is cleared.

**NOWAV** (Type: Statement Cat #: 092)
Line Range: 7012–7026
Modules: SOUND THINGS
Example: NOWAV 5
Clear the wave form nybble in the specified voice(s).

**NOI** (Type: Statement Cat #: 093)
Line Range: 6996–7026
Modules: SOUND THINGS
Example: NOI 1
Set the wave form to noise in the specified voice(s).

**PUL** (Type: Statement Cat #: 094)
Line Range: 7000–7026
Module: SOUND THINGS
Example: PUL 7
Set the wave form to pulse in the specified voice(s).

**SAW** (Type: Statement Cat #: 095)
Line Range: 7004–7026
Modules: SOUND THINGS
Example: SAW VV
Set the wave form to sawtooth in the specified voice(s).

**TRI** (Type: Statement Cat #: 096)
Line Range: 7008–7026
Modules: SOUND THINGS
Example: TRI V1+V2+V3
Set the wave form to triangular in the specified voice(s).

**TEST** (Type: Statement Cat #: 097)
Line Range: 7028–7052
Modules: SOUND THINGS
Example: TEST 2,1
Set or clear the test bit in the wave form register of the specified voice(s). The first parameter is the voice(s). The second is set (1) or clear (0).

**RING** (Type: Statement Cat #: 098)
Line Range: 7032–7052
Modules: SOUND THINGS
Example: RING B,0
Switch ring modulation off or on in the specified voice(s). The first parameter is the voice(s). The second is on (1) or off (0).

**SYNC** (Type: Statement Cat #: 099)
Line Range: 7036–7052
Module: SOUND THINGS
Example: SYNC 4,1
Switch synchronization off or on in the specified voice(s). The first parameter is the voice(s). The second is on (1) or off (0).

**GATE** (Type: Statement Cat #: 100)
Line Range: 7040–7052
Module: SOUND THINGS
Example: GATE 2,1
Set or clear the gate bit in the wave form register of the specified voice(s). The first parameter is the voice(s). The second is set (1) or clear (0). Setting the gate bit starts the attack phase of the ADSR envelope; clearing the gate bit start the release phase.

**ATT** (Type: Statement Cat #: 101)
Line Range: 7054–7070
Module: SOUND THINGS
Example: ATT 1,2
Set the attack time in the specified voices (first argument) to the value in the second argument (range 0–15).

**DEC** (Type: Statement Cat #: 102)
Line Range: 7072–7092
Module: SOUND THINGS
Example: DEC 6,11
Set the decay time in the specified voices (first argument) to the value in the second argument (range 0–15).

**SUS** (Type: Statement Cat #: 103)
Line Range: 7058–7070
Module: SOUND THINGS
Example: SUS 3,15
Set the sustain volume level in the specified voices (first argument) to the value in the second argument (range 0–15).

**REL** (Type: Statement Cat #: 104)
Line Range: 7076–7092
Module: SOUND THINGS
Example: REL 7,0
Set the release time in the specified voices (first argument) to the value in the second argument (range 0–15).

**RESON** (Type: Statement Cat #: 105)
Line Range: 7094–7112
Module: SOUND THINGS
Example: RESON 11
Set the filter resonance level to the specified value.

**VOL** (Type: Statement Cat #: 106)
Line Range: 7102–7112
Module: SOUND THINGS
Example: VOL 6
Set the combined volume level for the three SID voices to the specified value.

**FILT** (Type: Statement Cat #: 107)
Line Range: 7114–7124
Module: SOUND THINGS
Example: FILT 12,1
Switch the filter on or off. The first parameter is the voice(s) as usual, except that a fourth bit, corresponding to the audio input to the SID chip, is included. That bit contributes a value of 8 to the total for the voices selected. The second parameter in this statement is 1 (for on) or 0 (for off). Thus the example selects filtering on for the audio input and for the third SID voice.

**TRDOFF** (Type: Statement Cat #: 108)
Line Range: 7126–7138
Module: SOUND THINGS
Example: TRDOFF
Switches off oscillator 3.

**TRDON** (Type: Statement Cat #: 109)
Line Range: 7130–7138
Module: SOUND THINGS
Example: TRDON
Switches on oscillator 3.

**HP** (Type: Statement Cat #: 110)
Line Range: 7140–7158
Module: SOUND THINGS
Example: HP 1
Turn the high pass filter on or off, leaving the status of the other two filters unchanged. The parameter is 1 (on) or 0 (off).

**BP** (Type: Statement Cat #: 111)
Line Range: 7144–7158
Module: SOUND THINGS
Example: BP 0
Turn the band pass filter on or off, leaving the status of the other two filters unchanged. The parameter is 1 (on) or 0 (off).

**LP** (Type: Statement Cat #: 112)
Line Range: 7148–7158
Module: SOUND THINGS
Example: LP FS
Turn the low pass filter on or off, leaving the status of the other two filters unchanged. The parameter is 1 (on) or 0 (off).

**POTX** (Type: Function Cat #: 113)
Line Range: 7060–7178
Module: SOUND THINGS
Example: P=POTX
This pseudo-variable returns the value of a game paddle plugged into joystick port 1.

**POTY** (Type: Function Cat #: 114)
Line Range: 7064–7178
Module: SOUND THINGS
Example: PRINT POTY
This pseudo-variable returns the value of a game paddle plugged into joystick port 2.

**OSC3** (Type: Function Cat #: 115)
Line Range: 7068–7178
Module: SOUND THINGS
Example: J=OSC3*256
This pseudo-variable returns the current value of the upper 8 bits of the output of oscillator three.

**ENV3** (Type: Function Cat #: 116)
Line Range: 7072–7178
Module: SOUND THINGS
Example: FREQ 1,20000+ENV 3*10
This pseudo-variable returns the current value of the envelope generator of oscillator three.

## Modules So Far

TransBASIC Modules that have appeared so far (Instalments 1 to 4)

### TransBASIC #1

**TB/KERNEL**

Statements: 2 Functions: 0 Keyword Characters: 8

| | |
|---|---|
| 000 S/IF | Modified IF to work with TransBASIC |
| 001 S/ELSE | Part of IF–ELSE construct |
| 002 S/EXIT | Disable current TransBASIC dialect |

**SCREEN THINGS**

Statements: 5 Functions: 0 Keyword Characters: 22

| | |
|---|---|
| 013 S/GROUND | Set background colour |
| 014 S/FRAME | Set border colour |
| 015 S/TEXT | Set text colour |
| 016 S/CRAM | Fill colour memory with value |
| 017 S/CLS | Clear screen, or screen line range |

### TransBASIC #2

**DOKE & DEEK**

Statements: 1 Functions: 1 Keyword Characters: 9

| | |
|---|---|
| 007 S/DOKE | Poke a 16–bit value |
| 008 F/DEEK( | Peek a 16–bit value |

**BIT TWIDDLERS**

Statements: 3 Functions: 0 Keyword Characters: 12

| | |
|---|---|
| 009 S/SET | Set specified bit at address |
| 010 S/CLEAR | Clear specified bit at address |
| 011 S/FLIP | Flip specified bit at address |

**CHECK & AWAIT**

Statements: 0 Functions: 2 Keyword Characters: 12

| | |
|---|---|
| 018 F/CHECK( | Check keyboard for valid character |
| 019 F/AWAIT( | Wait for valid character from keyboard |

**KEYWORDS**

Statements: 1 Functions: 0 Keyword Characters: 8

| | |
|---|---|
| 059 S/KEYWORDS | Print currently active TransBASIC keywords |

### TransBASIC #3

**CURSOR POSITION**

Statements: 1 Functions: 1 Keyword Characters: 10

| | |
|---|---|
| 004 S/CURSOR | Move cursor to specified row and column |
| 005 F/CLOC | Return cursor location |

**SET SPRITES**

Statements: 6 Functions: 0 Keyword Characters: 27

| | |
|---|---|
| 031 S/COLSPR | Set colour of sprite |
| 032 S/SSPR | Turn on a sprite |
| 033 S/CSPR | Turn off a sprite |
| 034 S/XSPR | Move sprite to specified x–position |
| 035 S/YSPR | Move sprite to specified y–position |
| 036 S/XYSPR | Move sprite to specified xy–position |

**WITHIN**

Statements: 0 Functions: 1 Keyword Characters: 7

| | |
|---|---|
| 040 F/WITHIN( | Return true if value lies within specified range |

**READ SPRITES**

Statements: 0 Functions: 2 Keyword Characters: 10

| | |
|---|---|
| 041 F/XLOC( | Return x–position of specified sprite |
| 042 F/YLOC( | Return y–position of specified sprite |

### TransBASIC #4

**STRIP & CLEAN**

Statements: 0 Functions: 2 Keyword Characters: 14

| | |
|---|---|
| 045 F/STRIP$( | Remove non–alphanumerics from string |
| 046 F/CLEAN$( | Remove non–blank non–alphanumerics from string |

**SCROLLS**

Statements: 4 Functions: 0 Keyword Characters: 24

| | |
|---|---|
| 067 S/USCROL | Scroll screen area up one row |
| 068 S/DSCROL | Scroll screen area down one row |
| 069 S/LSCROL | Scroll screen area left one row |
| 070 S/RSCROL | Scroll screen area right one row |

**Editor's Note:** This jumbo TransBASIC article has been brought to you thanks to the diligent efforts of Nick Sullivan. Although several of the modules this time were submitted by readers, much work went into preparing them. As mentioned, Nick found it necessary to edit almost everything; in all cases the line numbers were modified; labels were changed in the source listings to cut down on the chances of duplicates; keywords had to be changed in many cases to eliminate tokenization problems (eg. 'RES' was one of the Sound Things keywords but had to be changed so as not to interfere with RESTORE); and commenting, general organization, not to mention the presentation itself, ate up some hours, I'm sure.

For those who submitted TransBASIC modules, The Transactor will be sending a free 1 year magazine subscription, plus the Transactor Disk for this issue (Disk #8) so you don't have to retype your own modules to resemble what Nick has done to them.

As promised last issue, the following is a quick refresher on building a TransBASIC dialect. M.Ed.

### Using TransBASIC

About the easiest way to get in on TransBASIC is to obtain a copy of The Transactor Disk (Disk #4 or greater). TransBASIC users must also have the PAL Assembler package (or a similar assembler as discussed earlier).

The directory shows a program called "transbasic instr". LOAD and LIST and you will see that it will proceed to load two other programs: the first is the 'ADD' module which allows you to add more modules to the 'tb/kernel' which is loaded second.

Now comes the easy part. Select the modules you need from those you have on disk (Disk #8 contains every module released to date). Then, for each module, follow these steps:

1) Use the ADD statement to merge the module into memory, for example:

```
ADD "SCREEN THINGS"
```

2) List line 2 of your program. This line number is common to all modules. It will read something like:

```
REM 5 STATEMENTS, 0 FUNCTIONS
```

3) List line 95. This kernel line records the number of statements and functions in the TransBASIC that you are creating. When you first load in the kernel, line 95 reads:

```
95 XTRA .BYTE 2,0   ; STMTS,FNCS
```

. . .indicating that the kernel contains two statements (ELSE and EXIT) and no functions. You are responsible for updating the two numbers appropriately as you ADD modules. After adding SCREEN THINGS, for instance, the first number in line 95 would be increased by five, the second would be left unchanged.

When you have finished adding modules, it would probably be a good idea to save the completed source file, at least temporarily. Then load PAL, if you haven't previously, and give the RUN command. PAL then proceeds to assemble all the modules you 'ADDed' into your new TransBASIC extension.

Normally the object code is origined to that popular niche at $C000, but you can select another starting point if you wish (see line 31 of the source code). Save the object code directly, perhaps with Supermon, or convert it into DATA statements that can be loaded in with whatever programs you intend shall make use of the added commands.

With that, the work is done. To activate the new commands type:

```
SYS 49152
```

Presto! — you have just extended BASIC to your own specifications, and now it's ready for use.

### Program 1: LABELS

```
JL  0 rem labels (j. gillaspie 3/85)     :
FH  1 :
MH  2 rem 5 statements, 0 functions
HH  3 :
KE  4 rem keyword characters: 24
JH  5 :
NJ  6 rem keyword     routine    line    ser #
JK  7 rem l.          ='data'    $adf8   073
HI  8 rem lgoto       lgot       5924    074
CI  9 rem lgosub      lgosu      5870    075
CL  10 rem sgoto      sgot       5920    076
KK  11 rem sgosub     sgosu      5866    077
AI  12 :
LD  13 rem ================================
CI  14 :
JF  120 .byte $4c,$ae: .asc "lgotOlgosuB"
OB  121 .asc "sgotOsgosuB"
PP  1120 .word $a8f7,lgot-1,lgosu-1
HE  1121 .word sgot-1,sgosu-1
DB  5866 sgosu sec
IL  5868       .byte $24
OP  5870 lgosu clc
JJ  5872       ror  t6        ;s=neg, l=pos
BO  5874       lda  #$ff      ;max string length
IF  5876       sta  t5
AE  5878       lda  #3        ;duplicate rom's
ML  5880       jsr  $a3fb     ; gosub routine
EP  5882       lda  $7b       ;push chrget ptr
IH  5884       pha
LL  5886       lda  $7a
MH  5888       pha
DN  5890       bit  t6        ;test jump-type flag
HE  5892       bpl  lgos1
II  5894       jsr  sgstr     ;evaluate string
ON  5896 lgos1 lda  $3a       ;push line number
GI  5898       pha
JK  5900       lda  $39
KI  5902       pha
MB  5904       lda  #$8d      ;push gosub token
OI  5906       pha
GC  5908       jsr  $79
CA  5910       dey            ;back up token offset
GI  5912       dey            ; to labelled goto
HH  5914       jsr  lgot1     ;use labelled goto
KK  5916       jmp  $a7ae     ;next statement
EJ  5918 ;
NB  5920 sgot  sec
OO  5922       .byte $24
BB  5924 lgot  clc
PM  5926       ror  t6        ;s=neg, l=pos
HB  5928       lda  #$ff      ;max string length
OI  5930       sta  t5
NP  5932       bit  t6        ;test jump-type flag
EH  5934       bpl  lgot1
```

```
CL  5936        jsr  sgstr      ;evaluate string
IJ  5938 lgot1  dey             ;back up token offset
JL  5940        dey             ; to l. command
PG  5942        tya             ;convert to token
CM  5944        lsr             ; stored in t4
FA  5946        ora  #$40
OJ  5948        sta  t4
FA  5950        cmp #$5d
PD  5952        bcc  lgot2
GI  5954        inc  t4
HC  5956 lgot2  lda  $7a        ;save chrget ptr
EK  5958        sta  t2
IA  5960        lda  $7b
KK  5962        sta  t3
MH  5964        lda  $2b        ;start of basic ptr
DG  5966        ldx  $2c
GL  5968        ldy  #1         ;point to link hi byte
LJ  5970 lgot3  sta  $5f        ;set zp pointer
FM  5972        stx  $60        ; to current line
DA  5974        lda  ($5f),y    ;check for end of pgm
OB  5976        beq  lgot10     ;yes, undef'd stmt
OB  5978        ldy  #4         ;point to 1st tok byte
CJ  5980        lda  ($5f),y    ;get it
CI  5982        cmp #$5f        ;check if tb token (←)
PP  5984        bne  lgot9      ;no, try next line
KE  5986        iny             ;yes
OD  5988        lda  ($5f),y    ;which tb token
OB  5990        cmp t4          ;check if label
HA  5992        bne  lgot9      ;no, try next line
JG  5994 lgot4  iny             ;strip off blanks
HI  5996        lda  ($5f),y
MN  5998        cmp #$20
FL  6000        beq  lgot4
DG  6002        ldx  t5         ;get string length
HA  6004        jsr  $79        ;begin label compare
EF  6006 lgot5  cmp ($5f),y
GJ  6008        bne  lgot9      ;no match, next line
GL  6010        iny             ;match, test next char
DD  6012        dex
LM  6014        beq  lgot6
AI  6016        jsr  $73
DN  6018        bne  lgot5      ;done if line/stmt end
KA  6020 lgot6  lda  ($5f),y;
MO  6022        beq  lgot7      ;yes, end of line
IG  6024        iny
GK  6026        cmp #$20        ;blanks don't count
JN  6028        beq  lgot6
JE  6030        dey
DG  6032        cmp # ":"       ;test end of stmt
AK  6034        bne  lgot9      ;no match
IM  6036 lgot7  lda  $5f        ;copy ptr to chrget
KH  6038        ldx  $60
OA  6040        clc
OO  6042        adc  #4         ;skip link, line #
DL  6044        bcc  lgot8
KH  6046        inx
MB  6048 lgot8  sta  $7a
MP  6050        stx  $7b
LE  6052        jmp  $a8f8      ;use data rtn to skip
BI  6054 lgot9  lda  t2         ;point back to
NE  6056        sta  $7a        ; start of label
MM  6058        lda  t3
KK  6060        sta  $7b
HI  6062        ldy  #1         ;point to link
PD  6064        lda  ($5f),y    ;to next line
NH  6066        tax
PG  6068        dey
GN  6070        lda  ($5f),y    ;get first char
IJ  6072        iny
JK  6074        bne  lgot3      ;look for next label
KO  6076 lgot10 bit  t6         ;test jump-type flag
BB  6078        bpl  lgot11     ;l-type, give up
IE  6080        clc             ;set flag to l-type
MP  6082        ror  t6
BE  6084        lda  #<trpstr   ;hunt 'dfault' label
BJ  6086        ldy  #>trpstr
DM  6088        sta  $7a
IC  6090        sty  $7b
DC  6092        ldy  #6
CJ  6094        sty  t5
BD  6096        jmp  lgot2
MC  6098 lgot11 jmp  $a8e3      ;undef stmt error
KI  6100 trpstr .asc "dfault"
OE  6102 sgstr  sty  $14        ;save token offset
JD  6104        jsr  $ad9e      ;eval label string
LI  6106        jsr  $b6a3      ;get strlen & addr
OK  6108        sta  t5         ;save length
NA  6110        stx  $7a        ;set chrget ptr
OH  6112        sty  $7b        ; to string data
NC  6114        ldy  $14        ;recover token offset
AN  6116        dey             ;back up token offset
IN  6118        dey             ; to labelled jump
DK  6120        dey
FK  6122        dey
GA  6124        sec             ;set s-jump flag
IC  6126        ror  t6
MN  6128        rts
IG  6130 ;
```

**Program 2: TOKEN & VAR**

```
CH  0 rem token & var (april 7/85)     :
FH  1 :
DH  2 rem 0 statements, 2 functions
HH  3 :
DE  4 rem keyword characters: 11
JH  5 :
NJ  6 rem keyword    routine   line   ser #
HB  7 rem token$(    token     6132   078
KC  8 rem var(       var       6198   079
NH  9 :
ME  10 rem u/usfp (2620/006)
PH  11 :
KD  12 rem ================================
BI  13 :
BK  611 .asc "token$" : .byte $a8
GD  612 .asc "var" : .byte $a8
EC  1611 .word token-1
FH  1612 .word var-1
IB  2620 usfp   ldx  #0         ;routine to convert
GM  2622        stx  $0d        ;unsigned integer
IN  2624        sta  $62        ;in .a (high byte)
OH  2626        sty  $63        ;and .y (low byte)
BB  2628        ldx  #$90       ;to floating point
KI  2630        sec             ; in fac #1
NH  2632        jmp  $bc49
AM  2634 ;
```

```
HO  6132 token jsr  $b3a6    ;program mode only
NC  6134       jsr  $aef4    ;evaluate expr
KP  6136       jsr  $b6a3    ;set string ptrs
EJ  6137       cmp #$59
LA  6138       bcs  tkn4     ;up to 88 chars
KM  6139       tay
CF  6140       lda  #0       ;clear .a and .x
JM  6142       tax
LO  6144 tkn1  sta  $200,y   ;copy string to
FA  6146       dey           ; input buffer
HM  6148       lda  ($22),y  ; with terminal 0
FD  6150       cpy  #$ff
CC  6152       bne  tkn1
BA  6154       lda  $7a      ;push chrget ptr
II  6156       pha
OM  6158       lda  $7b
MI  6160       pha
JG  6162       stx  $7a
OC  6164       jsr  tok      ;tokenize buffer
PB  6166       pla           ;pull chrget ptr
GB  6168       sta  $7b
CK  6170       pla
HB  6172       sta  $7a
PC  6174       tya           ;calc length of
PE  6176       sec           ; tokenized line
HC  6178       sbc  #5
OD  6180       jsr  $b47d    ;reserve str space
FP  6182       tay
AK  6184 tkn2  dey           ;copy tokenized
PP  6186       bmi  tkn3     ; line to string
DL  6188       lda  $200,y   ; storage
KD  6190       sta  ($62),y
OE  6192       bne  tkn2
MA  6194 tkn3  jmp  $b4ca    ;set up descriptor
DJ  6195 tkn4  jmp  $b658    ;string too long
KK  6196 ;
AJ  6198 var   jsr  $b08b    ;find variable
KP  6200       ldy  $47      ;load pointer
MG  6202       lda  $48      ; to data
NH  6204       jsr  usfp     ;conv to floating
GP  6206       jmp  $aef7    ;check for paren
GL  6208 ;
```

**Program 3: INSTRING**

```
NN  0 rem instring (c. kluepfel, apr/85)  :
FH  1 :
EC  2 rem 0 statements,  1 function
HH  3 :
GO  4 rem keyword characters: 6
JH  5 :
NJ  6 rem keyword     routine    line    ser #
LN  7 rem f/instr(    instr      6210    080
MH  8 :
HD  9 rem ===============================
OH  10 :
HB  613 .asc "instr" : .byte $a8
HD  1613 .word instr-1
HC  6210 instr  lda  #2       ;check stack depth
CL  6212        jsr  $a3fb
KH  6214        jsr  $ad9e    ;evaluate string 1
DN  6216        jsr  $b6a3    ; and set up ptrs
KK  6218        sta  t3
JC  6220        pha           ;push length
CA  6222        txa
AG  6224        pha           ;push addr-lo
JA  6226        tya
AE  6228        pha           ;push addr-hi
IH  6230        lda  t3
KI  6232        jsr  $b47d    ;lower b-o-s ptr
ME  6234        jsr  $aefd    ;check for comma
DJ  6236        jsr  $ad9e    ;evaluate string 2
JO  6238        jsr  $b6a3    ; and set up ptrs
KA  6240        stx  $22      ;store address ptr
JI  6242        sty  $23
LK  6244        sta  t3       ;store length
OO  6246        pla
CD  6248        sta  $25      ;set up addr ptr
JN  6250        pla           ; to string 1
GD  6252        sta  $24
GP  6254        pla
MD  6256        sta  t2       ;save length
BG  6258        sta  t4       ;set up test limit
DJ  6260        dec  t4
EJ  6262        ldx  #$af     ;'and' - default
KI  6264        jsr  $79
KL  6266        cmp  # ")"    ;branch on paren -
GF  6268        beq  ins1     ; end of expr
IA  6270        jsr  $aefd    ;test for comma
MA  6272        tax           ;boolean to .x
BO  6274        jsr  $73      ;get next char
MB  6276 ins1   jsr  $aef7    ;test for r. paren
HA  6278        sec
FA  6280        lda  t2       ;str1 null - exit
HN  6282        beq  ins6
CG  6284        sbc  t3       ;len str2-len str1
LM  6286        ror  t6       ;rot carry to t6
CL  6288        tay           ;result to .y
EL  6290        lda  t3
FM  6292        beq  ins6     ;str2 null - exit
BI  6294        lda  #0
JP  6296        sta  insctr   ;init counter
KO  6298        cpx  #$af     ; and ;test for and
JN  6300        beq  ins2
MJ  6302        cpx  #$b0     ; or ;test for or
BO  6304        beq  ins3
FK  6306        cpx  #$a8     ; not ;test for not
FO  6308        beq  ins3
MF  6310        jmp  $af08    ;syntax error
HM  6312 ins2   bit  t6       ;exit if len str2
BM  6314        bpl  ins6     ; > len str1
HP  6316        sty  t4       ;store test limit
OM  6318 ins3   ldy  #0       ;init index
CN  6320 ins4   lda  ($24),y  ;get str1 char
JH  6322        cpx  #$af     ;branch on or/not
IP  6324        bne  ins9
EG  6326        cmp  ($22),y  ;compare with str2
NK  6328        bne  ins7     ;skip if unequal
CF  6330        iny           ;advance index
PF  6332        cpy  t3       ;index = len str1
OA  6334        bne  ins4     ; means success
LB  6336 ins5   ldy  insctr   ;get function
DH  6338        .byte $2c     ; result (counter)
OM  6340 ins6   ldy  #$ff     ;make result zero
GK  6342        iny
LN  6344        jmp  $b3a2    ;result to fac 1
DP  6346 ins7   inc  insctr   ;bump counter
```

```
JI  6348          lda  t4          ;get test limit
DJ  6350          cmp inschr       ;branch if done
PN  6352          bcc  ins6
AP  6354          inc  $24         ;bump pointer
NH  6356          bne  ins8        ; into str1
FI  6358          inc  $25
AD  6360 ins8     bne  ins3        ;next pass
KA  6362 ins9     ldy  t3          ;get str1 len
FF  6364          cpx  #$a8        ;branch on 'not'
IE  6366          beq  ins11
OG  6368 ins10    dey              ;try to match any
MB  6370          cpy  #$ff        ; str2 char
BD  6372          beq  ins7        ;no, do next pass
NO  6374          cmp  ($22),y
MD  6376          bne  ins10       ;no, try next char
BH  6378          beq  ins5        ;yes, exit
MH  6380 ins11    dey              ;try to match any
IC  6382          cpy  #$ff        ; str2 char
NM  6384          beq  ins5        ;no, exit
JP  6386          cmp  ($22),y
JE  6388          bne  ins11       ;no, try next char
PA  6390          beq  ins7        ;yes, do next pass
OG  6392 ;
NF  6394 inschr   .byte 0          ;counter
CH  6396 ;
```

## Program 4: PLACE

```
GG  0 rem place (m. phillips 3/85)       :
FH  1 :
EC  2 rem 0 statements, 1 function
HH  3 :
GO  4 rem keyword characters: 6
JH  5 :
NJ  6 rem keyword      routine     line     ser#
GH  7 rem f/place(     nst         6398     081
MH  8 :
NL  9 rem ==============================
OH  10 :
EJ  614 .asc " place " : .byte $a8
FK  1614 .word nst-1
BG  6398 nst      lda  #2          ;check stack space
OG  6400          jsr  $a3fb
DC  6402          lda  #0          ;default start char
AI  6404          pha
HB  6406          jsr  $ad9e       ;evaluate expr
EC  6408          bit  $0d         ;test type
AN  6410          bmi  nst2        ;skip if string
NH  6412          jsr  $b7a1       ;conv to byte in .x
AA  6414          jsr  $aefd       ;check for comma
FG  6416          pla              ;substitute value
OD  6418          txa              ; in .x for default
FA  6420          bne  nst1        ;must be >0
CL  6422          jmp  $b248       ;illegal quantity
FO  6424 nst1     dex
OM  6426          txa
IJ  6428          pha
AM  6430          jsr  $ad9e       ;evaluate next expr
DN  6432 nst2     jsr  $b6a3       ;set up string ptrs
FK  6434          sta  t3          ;save str1 length
MB  6436          pha              ;push str1 length
DB  6438          txa              ;push str1 addr
EK  6440          pha
BO  6442          tya
IK  6444          pha
NK  6446          lda  t3          ;lower b-o-s ptr
JG  6448          jsr  $b47d
CB  6450          jsr  $79         ;retrieve separator
NJ  6452          jsr  $aefd       ;must be comma
IN  6454          jsr  $ad9e       ;evaluate next expr
EN  6456          jsr  $b6a3       ;set up string ptrs
EA  6457          tax
HA  6458          beq  nst6        ;str2 null
NJ  6459          sta  t4
OE  6460          jsr  $79
DF  6462          jsr  $aef7       ;check right paren
BO  6464          pla              ;retrieve str1 addr
FL  6466          sta  $25         ;store at $24/25
MM  6468          pla
AB  6470          sta  $24
AN  6472          pla
FB  6473          beq  nst6        ;str1 null
NM  6474          sta  t3          ;save str1 length
EN  6476          pla
GC  6478          sta  t2          ;save start pos'n
LB  6480          sta  t5          ;init result
JK  6482          lda  t3          ;start pos'n must
LC  6484          cmp  t2          ; be within str1
FL  6486          beq  nst6
JH  6488          bcc  nst6
EE  6490          sbc  t4          ;str1 cannot be
IK  6492          bcc  nst6        ; shorter than str2
BL  6494          sta  t6          ;save # of loops
EE  6496 nst3     clc              ;advance pointer to
PO  6498          lda  $24         ; str1, reflecting
BJ  6500          adc  t5          ; start position
AD  6502          sta  $24
BI  6504          bcc  nst4
JB  6506          inc  $25
EN  6508 nst4     ldy  #1          ;bump str1 ptr by 1
OE  6510          sty  t5          ; at nst3 next time
HP  6512          dey              ;index into str1
GN  6514 nst5     lda  ($24),y     ;get a character
DM  6516          cmp  ($22),y     ;branch if no match
MD  6518          bne  nst7        ; with str2
LB  6520          iny              ;bump index
NN  6522          cpy  t4          ;branch if more
KH  6524          bne  nst5        ; chars to test
AF  6526          ldy  t2          ;get result
AG  6528          iny
OJ  6530          .byte $2c        ;'bit' instruction
NA  6532 nst6     ldy  #0          ;search failed
NO  6534          jmp  $b3a2       ;result to fac #1
AK  6536 nst7     lda  t2          ;quit if no more
HD  6538          cmp  t6          ; positions to
AE  6540          bcs  nst6        ; search from
MB  6542          inc  t2          ;bump result
PA  6544          bne  nst3        ;try again
IA  6546 ;
```

### Program 5: ARCFUNCTIONS

```
EM  0 rem arcfunctions (c. kluepfel 3/85) :
FH  1 :
DH  2 rem 0 statements, 2 functions
HH  3 :
PH  4 rem keyword chars: 8
JH  5 :
NJ  6 rem keyword    routine   line   ser#
AH  7 rem f/asn(     asin      6548   082
EF  8 rem f/acs(     acos      6670   083
NH  9 :
ID  10 rem ===============================
PH  11 :
MD  615 .asc "asn" : .byte $a8
MC  616 .asc "acs" : .byte $a8
AF  1615 .word asin-1
OE  1616 .word acos-1
LP  6548 asin     lda  #2          ;test stack depth
EA  6550          jsr  $a3fb
IP  6552          jsr  $79         ;reexamine byte
KP  6554          jsr  $aef4       ;eval, right paren
HP  6556          jsr  $ad8d       ;check expr numeric
JC  6558          lda  $66         ;push sign
MB  6560          pha
BP  6562          lda  #0          ;make it positive
MH  6564          sta  $66
NH  6566          lda  #<$b9bc     ;point to number 1
PO  6568          ldy  #>$b9bc
PL  6570          jsr  $bc5b       ;compare with fac#1
AJ  6572          beq  asi1        ;branch if equal
IC  6574          bmi  asi3        ; or if fac is less
AA  6576          lda  #0          ;clear low byte
LC  6578          sta  $65         ; of mantissa
AF  6580          sta  $70         ; and rounding byte
FP  6582          lda  #<$b9bc     ;repeat comparison
PP  6584          ldy  #>$b9bc
GB  6586          jsr  $bc5b
JN  6588          beq  asi1
LN  6590          bmi  asi3
PL  6592          jmp  $b248       ;ill quant if >1
JI  6594 asi1     lda  #<$e2e0     ;point to pi/2
NN  6596          ldy  #>$e2e0
EL  6598          jsr  $bba2       ;copy to fac#1
DK  6600          pla              ;restore sign
HE  6602          sta  $66         ; and exit
JI  6604 asi2     rts
LH  6606 asi3     pla              ;restore sign
IK  6608          sta  $66
JC  6610          lda  $61         ;if argument is 0,
CO  6612          beq  asi2        ; so is result
DJ  6614          jsr  $bc1b       ;round fac#1
JM  6616          lda  #3          ;check stack space
IE  6618          jsr  $a3fb
NP  6620          ldx  #5          ;push fac#1
EE  6622 asi4     lda  $61,x
MF  6624          pha
JJ  6626          dex
KB  6628          bpl  asi4
CA  6630          jsr  $bc0c       ;copy fac to fac#2
CP  6632          jsr  flmult      ;square fac#1
BM  6634          lda  #<$b9bc     ;point to number 1
DD  6636          ldy  #>$b9bc
OO  6638          jsr  $b850       ;calc 1-(fac#1)
PM  6640          jsr  $bf71       ;calc sqr(fac#1)
JD  6642          ldx  #0
II  6644 asi5     pla              ;pull fac#2
PN  6646          sta  $69,x
EN  6648          inx
PF  6650          cpx  #6
EB  6652          bne  asi5
DB  6654          pha              ;push sign again
ON  6656          lda  $61         ;branch on
MB  6658          beq  asi1        ; zero result
MI  6660          pla
NM  6662          lda  $61         ;calc fac#2/fac#1
BA  6664          jsr  fldiv
KM  6666          jmp  $e30e       ;perform atn
CI  6668 ;
EH  6670 acos     jsr  asin        ;perform asin
HA  6672          lda  #<$e2e0     ;point to pi/2
LC  6674          ldy  #>$e2e0
KG  6676          jmp  $b850       ;calc pi/2 - fac#1
MI  6678 ;
GC  6680 flmult   jsr  condsg      ;multiply fac#1
JD  6682          jmp  $ba2b       ; by fac#2
CJ  6684 ;
BA  6686 fldiv    jsr  condsg      ;divide fac#2
NC  6688          jmp  $bb12       ; by fac#1
IJ  6690 ;
KP  6692 condsg   lda  $66         ;adjust sign
AB  6694          eor  $6e
AD  6696          sta  $6f
FL  6698          lda  $61
IB  6700          rts
EK  6702 ;
```

### Program 6: PRINTAT

```
AC  0 rem printat (s. erickson 3/85)   :
FH  1 :
AH  2 rem  1 statements, 0 functions
HH  3 :
GO  4 rem keyword characters: 6
JH  5 :
NJ  6 rem keyword    routine   line   ser #
FC  7 rem s/print@   prinat    6704   084
MH  8 :
HD  9 rem ===============================
OH  10 :
DP  122 .asc "print" : .byte $c0
KC  1122 .word prinat-1
DA  6704 prinat  jsr  $b79e     ;eval expr to .x
HL  6708         stx  $14       ;save (column #)
EB  6710         cpx  #$28      ;must be <40
JA  6712         bcs  prin1
MC  6714         jsr  $aefd     ;check for comma
HE  6720         jsr  $b79e     ;eval row to .x
JC  6722         cpx  #$19      ;must be <25
FB  6724         bcs  prin1
LB  6726         ldy  $14       ;column to .y
MD  6728         jsr  $fff0     ;kernal plot rtn
CI  6730         jsr  $79       ;quit if no
EO  6732         beq  prin2     ; string argument
LL  6734         jsr  $aefd     ;else check comma
FO  6736         jmp  $aaa0     ; & print string
PF  6738 prin1   jmp  $b248     ;illegal quantity
LF  6742 prin2   rts
OM  6744 ;
```

**Program 7: SOUND THINGS**

```
BG  0 rem sound things (f. vanzeist 3/85) :
FH  1 :
MB  2 rem 28 statements, 4 functions
HH  3 :
HO  4 rem keyword characters: 126
JH  5 :
MG  6 rem keywords #085 to #116          :
LH  7 :
BO  8 rem ==================================
NH  9 :
OP  123 .asc "clesiDfreQpuwiDfifreQ"
CM  124 .asc "adpuLadsaWadtrl"
BB  125 .asc "nowaVnolpuL"
DL  126 .asc "saWtrltesT"
FJ  127 .asc "rinGsynCgatE"
KF  128 .asc "atTdeCsuS"
EK  129 .asc "reLresoNvoLfilT"
OE  130 .asc "trdofFtrdoNhP"
BO  131 .asc "bPlP"
JF  617 .asc "potXpotY"
PH  618 .asc "osc" : .byte $b3 ;asc("3")+$80
BF  619 .asc "env" : .byte $b3
FG  1123 .word clesi-1,frq-1,puwi-1,fifre-1
NM  1124 .word adwav-1,adwv1-1,adwv2-1
JL  1125 .word nuwv4-1,nuwav-1,nuwv1-1
HI  1126 .word nuwv2-1,nuwv3-1,wavbit-1
AE  1127 .word wvbit1-1,wvbit2-1,wvbit3-1
PP  1128 .word asset-1,drset-1,ast1-1
PH  1129 .word drt1-1,rvset-1,rvt1-1,filt-1
LA  1130 .word third-1,thrd1-1,flset-1
JA  1131 .word flt1-1,flt2-1
BE  1617 .word pots-1,pts1-1
KL  1618 .word pts2-1
ML  1619 .word pts3-1
FH  6746 getvoi  jsr  $b79e      ;get byte in .x
CE  6748         cpx  #8         ;maximum 7 for
CN  6750         bcs  illqty     ;voice parameter
FD  6752         stx  voictr
OE  6754         rts
KN  6756 ;
KM  6758 getwrd  jsr  $aefd      ;check comma
PG  6760         jsr  $ad8a      ;get two bytes
CJ  6762         jsr  $b7f7      ;convert to int
ML  6764         lda  #<direct   ;address of direct
KB  6766         sta  sbyt3+1    ;routine replaces
NK  6768         lda  #>direct   ;dummy in sbyt1
PE  6770         sta  sbyt3+2    ;subroutine
AG  6772         rts
MO  6774 ;
KP  6776 lonyb   jsr  $aefd      ;check comma
FD  6778 lnyb1   jsr  $b79e      ;get byte in .x
ON  6780         cpx  #$10       ;maximum value of
AD  6782         bcs  illqty     ;one nybble is 15
MG  6784         rts
IP  6786 ;
OC  6788 hinyb   jsr  $aefd      ;check comma
HO  6790 hnyb1   jsr  lnyb1      ;get nybble
MD  6792         txa
GH  6794         asl             ;convert to
JJ  6796         asl             ;high nybble
JD  6798         asl
LD  6800         asl
NF  6802         tax
AI  6804         rts
MA  6806 ;
JK  6808 getbit  jsr  $aefd      ;check comma
JP  6810 gbit1   jsr  $b79e      ;get byte in .x
AF  6812         cpx  #0         ;must be 1 or 0
BG  6814         bne  gbit2      ;if .x is 0 then
IJ  6816         stx  newval     ;clear newval
PA  6818 gbit2   cpx  #2
ME  6820         bcs  illqty
CJ  6822         rts
OB  6824 ;
BN  6826 direct  lda  $14        ;direct pokes a
PP  6828         sta  imsid,y    ;two byte number
NE  6830         sta  $d400,y    ;for frequency,
CC  6832         lda  $15        ;pulsewidth and
KD  6834         sta  imsid+1,y  ;filter cutoff
OC  6836         sta  $d401,y    ;frequency
CK  6838         rts
OC  6840 ;
JN  6842 bitnyb  lda  imsid,y    ;set and clear
JH  6844         and  prtect     ;bit in sid
DL  6846         ora  newval     ;and imsid
DD  6848         sta  imsid,y    ;registers
LK  6850         sta  $d400,y    ;depending on
OH  6852         rts             ;newval
MD  6854 ;
CO  6856 illqty  jmp  $b248      ;ill quant error
AE  6858 ;
KC  6860 sidbyt  lda  #<bitnyb   ;set up to
OC  6862         sta  sbyt3+1    ;enter parameters.
ON  6864         lda  #>bitnyb   ;put bitnyb instead
AK  6866         sta  sbyt3+2    ;of dummy
DK  6868 sbyt1   sty  voindx     ;reg offset
JA  6870         ldx  #3         ;loop counter
PO  6872 sbyt2   lsr  voictr     ;check voice
NL  6874         bcc  sbyt4      ;don't change voice
DC  6876         ldy  voindx     ;get reg. offset
GD  6878 sbyt3   jsr  $0000      ;direct or bitnyb
MD  6880 sbyt4   lda  voindx
FI  6882         clc             ;add 7 to register
HH  6884         adc  #7         ;offset for next
OE  6886         sta  voindx     ;voice
PJ  6888         dex
HK  6890         bne  sbyt2      ;do another voice
IN  6892         rts
EG  6894 ;
CA  6896 eormsk  txa             ;.a is #$ff
ID  6898         eor  #$ff       ;complement of .x
CJ  6900 emsk1   stx  newval
LI  6902         sta  prtect
EO  6904         rts
AH  6906 ;
KF  6908 clesi   ldy  #$19       ;clears sid chip
MB  6910         lda  #0         ;and its image
BF  6912 csid1   sta  imsid,y
DE  6914         sta  $d400,y
PL  6916         dey
PK  6918         bpl  csid1
EP  6920         rts
AI  6922 ;
DE  6924 frq     jsr  getvoi     ; frequency
DO  6926         jsr  getwrd     ;get voice(s) and
CB  6928         ldy  #0         ;frequency, reg 0
IF  6930         jmp  sbyt1      ;enter frequency
KI  6932 ;
```

```
BG 6934 puwi    jsr  getvoi     ; pulse width
NO 6936         jsr  getwrd     ;get voice(s) and
PH 6938         lda  $15        ;pulse width
JE 6940         cmp #$10        ;maximum $0fff
GM 6942         bcs  illqty
OP 6944         ldy  #2         ;register 2
IB 6946         jmp  sbyt1      ;enter pulse width
KJ 6948 ;
EL 6950 fifre   jsr  $ad8a      ;cutoff frequency
EL 6952         jsr  $b7f7      ;conv to integer
BH 6954         ldx  #0
DD 6956 ffre1   asl  $14        ;rotate 5 bits of
HC 6958         rol  $15        ;lo byte into hi
NG 6960         bcs  illqty     ;maximum  $07ff
OA 6962         inx
HJ 6964         cpx  #5
FK 6966         bne  ffre1      ;another bit to go
BH 6968 ffre2   lsr  $14        ;put the 3 bits in
ME 6970         dex             ;lsb back in their
IO 6972         bne  ffre2      ;proper position
FH 6974         ldy  #$15       ;reg. 24 , filter
GF 6976         jmp  direct     ;cutoff frequency
IL 6978 ;
CB 6980 adwav   ldx  #$40       ;add pulse
BC 6982         .byte $2c
AF 6984 adwv1   ldx  #$20       ;add sawtooth
FC 6986         .byte $2c
IB 6988 adwv2   ldx  #$10       ;add triangle
KL 6990         lda  #$7f       ;protect whole reg
FL 6992         bne  gowave     ;except noise
IM 6994 ;
HK 6996 nuwav   ldx  #$80       ;set noise
BD 6998         .byte $2c
II 7000 nuwv1   ldx  #$40       ;set pulse
FD 7002         .byte $2c
NP 7004 nuwv2   ldx  #$20       ;set sawtooth
JD 7006         .byte $2c
FM 7008 nuwv3   ldx  #$10       ;set triangle
ND 7010         .byte $2c
OB 7012 nuwv4   ldx  #0         ;clear waveform
BO 7014         lda  #$0f
ON 7016 ;
PM 7018 gowave  jsr  emsk1      ;store values
ND 7020         jsr  getvoi     ;get voice(s)
IF 7022         ldy  #4         ;register 4
BM 7024         jmp  sidbyt     ;enter waveform
IO 7026 ;
MM 7028 wavbit  ldx  #8         ;test
BF 7030         .byte $2c
ME 7032 wvbit1  ldx  #4         ;ring modulation
FF 7034         .byte $2c
ML 7036 wvbit2  ldx  #2         ;synchronization
JF 7038         .byte $2c
ED 7040 wvbit3  ldx  #1         ;gate
KH 7042         jsr  eormsk
FF 7044         jsr  getvoi     ;get voice(s)
AE 7046         jsr  getbit     ;off or on
LN 7048         ldy  #4
GN 7050         jmp  sidbyt     ;enter parameter
CA 7052 ;
CC 7054 asset   ldy  #5         ;attack
LG 7056         .byte $2c
CM 7058 ast1    ldy  #6         ;sustain
LD 7060         sty  voindx     ;for indexed addr.
HG 7062         jsr  getvoi     ;get voice(s)
GG 7064         jsr  hinyb      ;get att/sus value
JF 7066         lda  #$0f       ;protect decay &
AC 7068         bne  drt2       ;release nybble
EB 7070 ;
JG 7072 drset   ldy  #5         ;decay
NH 7074         .byte $2c
AI 7076 drt1    ldy  #6         ;release
NE 7078         sty  voindx     ;for indexed addr.
JH 7080         jsr  getvoi     ;get voice(s)
NO 7082         jsr  lonyb      ;get dec/rel value
EK 7084         lda  #$f0       ;protect att/sus
IL 7086 drt2    jsr  emsk1
DH 7088         ldy  voindx
DI 7090         jmp  sidbyt     ;enter values
KC 7092 ;
BN 7094 rvset   jsr  hnyb1      ;resonance
CH 7096         ldy  #$17       ;register 23
PC 7098         lda  #$0f       ;protect lo nybble
AA 7100         bne  rvt2
LI 7102 rvt1    jsr  lnyb1      ;volume
BI 7104         ldy  #$18       ;register 24
BB 7106         lda  #$f0       ;protect hi nybble
GP 7108 rvt2    jsr  emsk1
CM 7110         jmp  bitnyb     ;enter values
OD 7112 ;
NC 7114 filt    jsr  lnyb1      ;filter
EM 7116         jsr  eormsk
OM 7118         jsr  getbit     ;get off or on
KI 7120         ldy  #$17       ;register 23
OM 7122         jmp  bitnyb     ;enter values
KE 7124 ;
AH 7126 third   ldx  #$80       ;third voice off
DL 7128         .byte $2c
ON 7130 thrd1   ldx  #0         ;third voice on
BF 7132         lda  #$7f       ;protect low bits
PJ 7134         ldy  #$18       ;register 24
EC 7136         bne  rvt2
IF 7138 ;
CM 7140 flset   ldx  #$40       ;high pass filter
BM 7142         .byte $2c
FK 7144 flt1    ldx  #$20       ;band pass filter
FM 7146         .byte $2c
II 7148 flt2    ldx  #$10       ;low pass filter
GO 7150         jsr  eormsk
FO 7152         jsr  gbit1      ;skip check comma
DL 7154         ldy  #$18       ;register 24
EK 7156         jmp  bitnyb     ;enter value
MG 7158 ;
KI 7160 pots    ldx  #0         ;potx reg offset
FN 7162         .byte $2c
MH 7164 pts1    ldx  #1         ;poty reg offset
JN 7166         .byte $2c
PO 7168 pts2    ldx  #2         ;osc3 reg offset
NN 7170         .byte $2c
PO 7172 pts3    ldx  #3         ;env3 reg offset
PC 7174         ldy  $d419,x    ;get value in reg.
NA 7176         jmp  $b3a2      ;store to fac #1
AI 7178 ;
HN 7180 imsid   *=*+$19
NB 7182 newval  *=*+1
JD 7184 prtect  *=*+1
JC 7186 voindx  *=*+1
OC 7188 voictr  *=*+1
MI 7190 ;
```

# The Atari 520ST An Overview

**Dave Gzik**
**Burlington, Ontario**

*This overview should in no way convey any indication that The Transactor is starting coverage of Atari computers. We fully intend to remain a Commodore exclusive journal, at least for the foreseeable future. Quite simply, we were interested in the information presented here and thought you might be too. M.Ed.*

For the past year or so Apple has been making inroads into the business market with a computer so easy to use, all you have to do is point and click.

Well up to now they have had no competition to speak of against the MacIntosh computer. Atari offers the solution to the people who dreamed of owning a Mac but were discouraged at the hefty price tag attached to it.

Presenting. . . the Atari 520 ST! Comparable in every way to the Mac except the price.

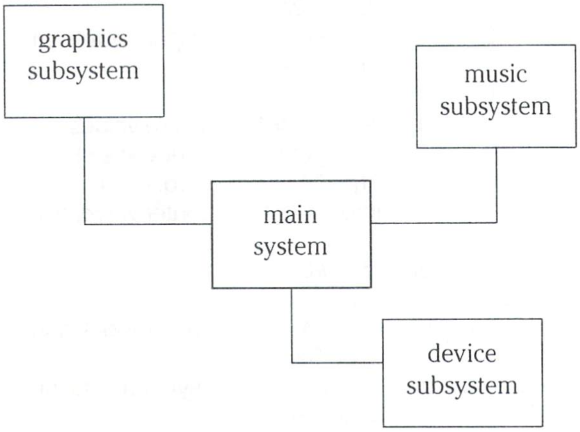
The following will give you some idea of the features the 520 ST has to offer you.

### Facts & Figures

The 520 ST computer is a GEM (Graphics Environment Manager) based 16/32 bit computer system that can facilitate many requirements for business, education, home, and specialty purposes.

The TOS operating system supports user interaction via a mouse controller to perform operations. These operations are shown on screen by ICONS which are graphic representations of operating system functions. Drop down menus and windows allow for easier identification of an operation to be selected.

The 520 ST is comprised of four systems which make up its architecture. The four systems are:




### Main System

The 520 ST computer is based on the Motorola 8 MHz 16 bit data/24 bit address microprocessor unit with an internal 32 bit architecture. This processor features eight 32 bit data registers, nine 32 bit address registers, a 16 megabyte direct addressing range, 14 addressing modes, memory mapped I/O (input/output), five data types, and a 56 mnemonic instruction set.

The main system contains 16 Kbytes of internal ROM (Read Only Memory) that contains the boot program for the operating system. The unit can accommodate an additional 128 Kbytes of ROM in cartridge form.

There are 512 Kbytes of RAM (Random Access Memory) on board and available on power up.

The main system also supports a direct memory access port that allows data transmission at a rate of 1.33 megabytes/second. This port will also serve as the Hard Disk interface.

**Graphics Subsystem**

The graphics subsystem of the ST possesses three modes of video configuration: 320 by 200 resolution with 4 planes, 640 by 200 resolution with 2 planes, and 640 by 400 resolution with 1 plane. (a plane represents the square number of colour palettes available) A sixteen word colour lookup palette is provided with nine bits of colour per entry. The sixteen colour palette registers contain three bits of red, green, and blue aligned on low nibble boundaries. Eight levels of red, green, and blue provide 512 maximum possible colours.

In low resolution 4 plane mode, all 16 palette colours are available, while in medium resolution 2 plane mode only the first four palette entries are accessible. In high resolution 1 plane mode the colour palette is bypassed altogether and is provided with an inverter for inverse video. Either the bit is on (white) or off (black).

The video display area uses 32 Kbytes that is mapped directly into RAM and has an identical bit, byte, and word relationship with the physical screen display.

**Music Subsystem**

The Atari ST Programmable Sound Generator (PSG) produces music synthesis, sound effects, and audio feedback. With an applied clock input of 2 MHz, this system is capable of producing frequency response from 30 Hz to 125 KHz. The sound system supports 3 voices with programmable envelope generator registers. The PSG three sound channel output is mixed together and sent out in a non amplified signal that can be received by a television, monitor speaker, or other amplifier devices. (The PSG has built–in digital to analog converters).

The Musical Instrument Digital Interface (MIDI) ports allow the ST to integrate with music synthesizers, sequencers, drum boxes, and other devices that support the MIDI interface. High speed (31.25 Kbaud) serial communications of keyboard and program information is provided by two ports, MIDI OUT and MIDI IN.

The MIDI bus permits up to a maximum of 16 channels in one of three addressing modes. OMNI mode allows all units addressed at once, POLY mode allows each unit addressed individually, and MONO which allows each unit voice addressed individually. MIDI information is communicated by five types of data along five data lines.

**Device Subsystem**

The device subsystem provides access to the ST via an intelligent keyboard (separate microprocessor controlled), and a two button mouse controller. The available ports for Input/Output on the ST are:

- 2 'D' style controller ports
- MIDI IN / MIDI OUT
- RGB/Monochrome monitor signal output
- Centronics Parallel
- RS–232 Serial
- Floppy Disk Serial
- Direct Memory Access/Hard Disk interface
- Direct Memory Expansion (ROM)

The monitor display port provides signal lines for either low resolution RGB, medium resolution RGB, or high resolution monochrome output.

A Standard Centronics Parallel port provides the ability to interface any compatible device directly to the ST without conversion interfaces. The ST RS–232 interface provides voltage level synchronous or asynchronous serial communication. The five standard RS–232C handshake control signals are supported allowing any compatible device to be connected without conversion interfaces. The ST RS–232 can support data transfer rates from 50 baud to 19.2 Kbaud.

The floppy disk port is setup to support ATARI three and half inch disk drives. Communication is achieved in a serial fashion through an Atari designed serial interface cable. The Hard Disk port supports a dual function. This port allows direct memory access (DMA) at 1.33 Mbytes/second. The communication method is parallel with a high speed throughput. Both disk ports contain on board controllers for their respective components.

The expansion port allows adding an additional 128 Kbytes of ROM. This cartridge based ROM can be utilized for application software, plug in languages, or as additional operating system information.

Well, that should be enough to digest for now. The newest Atari is the 520 ST available to consumers at a price that is one third that of the Mac. The 520 ST is packaged with a three and half inch microfloppy drive and a twelve-inch monochrome high resolution monitor. Also part of the package is the mouse controller, LOGO, BASIC, and the TOS operating system disk.

# Doing Away With Drama

Chris Zamara, Technical Editor

The second-rate actor staggers across the stage in his big death scene, gesticulating and gasping while taking out every obstacle in his path. This melodramatic spectacle is such a cliché that the only time you'll ever see it on stage or screen is probably as a parody. Why, then, is the computer-equivalent scene being played by almost every commercial software package on the market?

When you try to exit a program and go back to good ol' BASIC, why must you be subjected to colour flashes, cleared screens, and a cold restart? That's what you'll get with most word-processors, games, etc, providing they even have some means of exiting. Many don't. Turning a computer OFF then ON again just to try out something in BASIC or load in a new program (or to escape from the depths of some relentless mode!) is just a bit too vulgar to take. Like the over-achieving actor knocking down stage props, both of these escape options also tend to kill any data (or at least kill vital pointers) which have the misfortune of living in RAM at the time of program-abort. Due to the snail-like haste of the 1541 drive and hence the memory-intensive nature of most C-64 software, a cold start can leave you very cold indeed.

By insisting on taking complete control of the machine and cold-starting on exit, a program makes life much more difficult for itself than it has to. A program in that position assumes a lot of responsibility and becomes inadequate unless it gives the user options for his every whim — display disk catalog, allow sending of disk commands, provide a calculator or expression evaluator mode, etc. Otherwise, you get the dying-of-thirst-in-the-middle-of-the-ocean syndrome, sitting in front of your perfectly good computer, but not being able to calculate anything mathematical because you happen to be running a word-processor at the time.

I may be an incorrigible programmer at heart, but the only packages that get much use on my system are ones that I wrote myself, or ones written by other programmers, that don't give me extra drama for the money. Consider the terminal program for the 8032 that I use. It doesn't have a disk catalog function, but I don't care, because when I select the "Exit to BASIC" function, it simply says READY. That's it, no flashing, beeping, memory-clearing, or leaving a trail of broken props before exiting the stage. Now I can type CATALOG, do a calculation, or just play around in BASIC direct mode until I type RUN again to re-start the terminal program. I'm still connected with the host computer, and no drama distracts me from the task at hand. Give me a terminal program with a million extra features, and I don't want it unless it gives me elegant, non-destructive entry and exit. (While the argument that program exits must be destructive for software protection reasons could be brought up here, I think protection is even worse than memory-clearing. But that's another editorial.)

Unfortunately, program exits aren't the only over-dramatized event in software operation. Program entry or start-up is just as bad. How do the programmers dare to assume how I like my border, background, and character colours? I can set them up perfectly well myself, thank you. Changing colours is forgivable on some packages like games, but how about something like a disk copy utility? Why should you have to re-set all your colours after copying a few files just because some programmer somewhere liked pink letters on a green screen? (Doesn't matter, if he was like most programmers, the copy utility will probably cold-start after it's finished anyway, treating you to Commodore's wonderful blue-on-blue motif.)

You're probably saying to yourself, "Well what does this whining idiot want, anyway? A computer can only run one program at a time." Well, if you are, stop insulting me and I'll tell you. Having dabbled outside of the world of Commodore, I've seen some well-written (and expensive) packages running on IBM PCs. Dbase II is a good example — an incredibly powerful database management system with its own high-level language. You would expect such a system to completely take over the PC, but on start-up, it doesn't even clear the screen. When you bring it in (by simply typing "DBASE" from PC-DOS), the prompt just comes back in about a second, and the only clue that you're now in the Dbase command language instead of the operating system is the appearance of the prompt; a period instead of a greater-than. If Dbase ever falls short in the system command department, eg. examining disk files, just type QUIT and you get the PC-DOS prompt back. No files or data are lost, everything is saved, and Dbase retains its composure as it dies, much like an unwary victim succumbing to Mr. Spock's mysterious Vulcan grip. You can even automatically invoke Dbase from a batch file and exit again. The lack of drama here seems stark, but ah, so elegant! And so powerful!

As a computer-idealist, I look forward to the day when I can just call in programs one by one, flitting from terminal emulator to word-processor without any jolts to my sense of elegance. Programs which greedily change system parameters and vectors to the point where the only way back to normal is a cold start have no place in my computer-utopia. Programs must learn to live at peace with their environment as well as themselves. Since a computer cold-start is the equivalent of a nuclear holocaust on earth which wipes out all life, it's obvious that most commercial software hasn't learned yet. Like the melodramatic actor in his big scene, the dramatic program is somewhat embarrassing and awkward, as well as being a hindrance to the whole production. A change in direction is obviously needed here; let's not put software authors in the same company as bad actors.

# C Power – A Users Review

Richard Evers, Editor

*'C' makes you work to learn, but rewards you generously. . .*

C Power: It seems like a rather odd name for a software package. But if you can get past the stigma of its odd calling card, you will have discovered a friend for life. Written by Brian Hilchie, and distributed by Pro–Line Software, C Power is a C Language Editor/Compiler System for the Commodore 64. With that quick introduction out of the way, a little bit of C trivia is in order.

The C Language seems to be getting alot of air play these days. Major movies are programming their special effects in C (Star Wars, Star Trek), major software developers are writing their code in C (Micro-Soft, Visicorp), and simply put, it seems to the language of the future. Most of the Universities have been bitten by the C bug, with University and College students everywhere communicating in C. It's kind of like Valley speak, with class.

C Power allows the Commodore 64 user to write and compile in C. A simple statement to make, but not so simple when you get down to it. Unlike so many languages, C's secrets do not magically unravel with little effort. C makes you work to learn, but rewards you generously when you succeed. The true power of C lies in its relative simplicity, which seems to be anything but the truth at first glance. As time goes by, your awkward attempts at writing in C will start to pan out. But don't blame it entirely on the language. Learning a new language and a new system all at the same time can be rather frustrating. Time and perseverance seem to be the only way to conquer the first time blues.

The complete package as supplied by Pro–Line comes with one C Power diskette, one users guide, and one terrific book, C Primer Plus. The price for the package is $129.95 Canadian or $99.95 US.

The diskette supplied is a novelty. It is on the standard 1541/MSD format, but the trick is that both sides are used. In total, about 173 files are included on this disk. As stated in the users guide, only the compiler is copy protected. Everything else can be copied, and should be if you intend to actually use it.

When I first started writing this review, difficulties arose regarding the users manual. In simple terms, it was awful. Although it did contain some critical information deep within, it also had problems. Sections were missing, references to wrong pages were in plenty, and the presentation was poor. In despair, I called up Pro–Line and asked them if a better manual had been written. It turns out that my copy of the program was ancient (2 months). A new and improved 3rd printing had been made of the manual, and a super improved version of the program disk had also come about. Needless to say, my C Power misgivings were laid to rest. C Power became worthy of a review.

### Into The Unknown

The C Language, as stated earlier, will not welcome you with open arms. More than likely it will try to ignore you and hope you disappear. To get acceptance into the C club, some heavy duty reading and computer bashing will be required. The book, C Primer Plus, as supplied with the C Power package, is the ticket required to start to understand C, if you have the perseverance. Within its 500 pages plus, beginning to advanced concepts of programming in C are discussed. The authors went out of their way to bring the reader up through the ranks of C programming, in as short of time as possible. There is only one problem with the book. It has been written with the UNIX operating system in mind, with allowances for the MS DOS and CP/M–86 environments. The Commodore 64 shares little with any of these systems. It is simply not a UNIX type machine, therefore a few C concepts covered in the book are not applicable to the Commodore 64. All non–applicable sections and operations are discussed briefly in the C Power users guide.

Once you have stuck your nose in the C Primer for a short while, it would be best if you actually tried out the C Power package. Before doing so, read the users manual front to back. Unlike normal software packages, it expects you to know what you are doing. In order for you to generate true object of C code, you have to go through at least three separate stages with the system. The first is the editor, similar to a wordprocessor in the functions it performs. Once the editing work is completed, ie. you have written your code, a syntax checker is available for use. If your syntax is out in any way, this little beauty will pick it up and let you know. A nice touch.

Once you are satisfied that everything will be just right, the compiler lies in wait. As stated earlier, the compiler is the only program on disk that is protected. This is rather unfortunate, but is also a fact of life to live with as long as there are package pirates lurking about.

To continue, the compiler is a dream once you get it going. Even with the limitations of the 1541, it's not too slow. Also, as it compiles, you are able to see the source, pulling in the library routines as it goes along. A pretty impressive treat.

Once the compilation is complete, one more stage is required before you can call it executable object. You have to link all the code together. This means that you have to place your code plus the applicable library routines together to make one cohesive unit. The linker makes this part quite simple. If you want to make your code run in conjunction with the shell program supplied, the linker will take care of it. If you want true object that will run independently, this can also be arranged. Your code can be placed anywhere you want in memory, or can execute at the start of Basic, along with an applicable Basic line – SYS statement to get it going. It seems like quite a few stages to go through for object, but it really is worth it. It is true 6502 object, not P code.

When writing in C using C Power, you will probably notice a strange happening. The execution time of your code will vary depending on how you write your source. The C compiler supplied is not an optimizing compiler, therefore, if you do not plan your program properly, redundant code will be the result. The only cure for this is to become fluent in C, and the concepts behind it. Read the C Primer, work with the system, and if your head is screwed on properly, good clean code will be the result. Remember, becoming fluent in C could open many doors in the future.

### In Conclusion

In my opinion, the C Power system is a worthy investment. It may not be as fun as Comal, or as widely known as Basic, but it has more power than most realise. Due to this implementation of C by Brian Hilche, the source that you write on your 64 could be adapted to virtually any computer system supporting a C compiler, without major problems. Although the Commodore 64 does not allow for a true implementation of C, it's close enough to produce virtually transportable source. Without further argument, C Power makes the grade.

# COMMODORE 128 - Keywords and Tokens

Jim Butterfield
Toronto, Ontario

When the Commodore 128 is in the "64" mode, it behaves exactly the same as a 64 . . . in a sense, it is a 64. But when you select "128" mode, you have a new machine with much richer Basic. A good part of the machine is still familiar from the world of 64 – things such as POKE53281,0 still set the background color of the machine, for example. But Basic takes on a new, upward-compatible, set of keywords.

The average programmer may not care that keywords are changed into single-byte "tokens". In other words, a keyword such as INPUT is stored within the computer's memory as a numeric value of 133 – one byte represents the whole word. When you say LIST, the token is unfolded so that you see the original keyword.

The fact that each keyword has a specific token makes it convenient to give the keywords as a list. But there's a more important question: that of compatibility. If you have a program from a PET or a B-128 computer, it may have the right keywords, but the wrong token. As an example: if you use the command SCRATCH within a program on a PET 4.0 machine, the command will be stored (in memory or on a disk PRG file) as a value of 217 (hexadecimal D9). If you should load this program into the Commodore 128, the token comes in unchanged . . . but in the new machine, 217 stands for the keyword TROFF (trace off). The keyword SCRATCH exists in the 128, but it has a token value of 242 (hex F2).

This means that you may take a perfectly good PET/CBM 4.0 program, load it into the Commodore 128, and get nonsense. There are ways around this problem, but the first step is to know it can happen, and watch for it. By the way, this can't happen with programs being transferred from the Commodore 64 to the 128, since there is "upward compatibility". But if you go the other way, loading a 128 program which uses advanced commands into the 64 (or into a 128 in 64 mode), you'll see strange things in the program listing.

This keyword list allows me to comment briefly on the various keywords as they appear. This isn't a complete manual, but may help you place the new commands.

Key values are given in hexadecimal only. Advanced readers will notice that "double byte" tokens are used; this, too, is new. The double byte – the first byte always set to $FE or decimal 254 – also allows you to implement your own keywords if you wish.

**Fully 64 compatible:**

| | | | |
|---|---|---|---|
| 80 | END | A6 | SPC( |
| 81 | FOR | A7 | THEN |
| 82 | NEXT | A8 | NOT |
| 83 | DATA | A9 | STEP |
| 84 | INPUT# | AA | + |
| 85 | INPUT | AB | – |
| 86 | DIM | AC | * |
| 87 | READ | AD | / |
| 88 | LET | AE | (POWER) |
| 89 | GOTO | AF | AND |
| 8A | RUN | B0 | OR |
| 8B | IF | B1 | > |
| 8C | RESTORE | B2 | = |
| 8D | GOSUB | B3 | < |
| 8E | RETURN | B4 | SGN |
| 8F | REM | B5 | INT |
| 90 | STOP | B6 | ABS |
| 91 | ON | B7 | USR |
| 92 | WAIT | B8 | FRE |
| 93 | LOAD | B9 | POS |
| 94 | SAVE | BA | SQR |
| 95 | VERIFY | BB | RND |
| 96 | DEF | BC | LOG |
| 97 | POKE | BD | EXP |
| 98 | PRINT# | BE | COS |
| 99 | PRINT | BF | SIN |
| 9A | CONT | C0 | TAN |
| 9B | LIST | C1 | ATN |
| 9C | CLR | C2 | PEEK |
| 9D | CMD | C3 | LEN |
| 9E | SYS | C4 | STR$ |
| 9F | OPEN | C5 | VAL |
| A0 | CLOSE | C6 | ASC |
| A1 | GET | C7 | CHR$ |
| A2 | NEW | C8 | LEFT$ |
| A3 | TAB( | C9 | RIGHT$ |
| A4 | TO | CA | MID$ |
| A5 | FN | CB | GO |

**New functions:**

| | | | |
|---|---|---|---|
| CC | | RGR() | – return graphics mode |
| CD | | RCLR() | – return color value |
| CE | 02 | POT | – return selected paddle value |
| CE | 03 | BUMP | – return sprite collision data |
| CE | 04 | PEN | – return light pen coordinates |
| CE | 05 | RSPPOS | – return sprite speed & position |
| CE | 06 | RSPRITE | – return sprite characteristics |
| CE | 07 | RSPCOLOR | – return sprite multicolor values |
| CE | 08 | XOR | – return exclusive OR |
| CE | 09 | RWINDOW | – return size of window |
| CE | 0A | POINTER | – return address of variable |
| CF | | JOY() | – return joystick status |
| D0 | | RDOT() | – return values of pixel cursor |
| D1 | | DEC() | – return decimal value of hex string |
| D2 | | HEX$() | – return hex string |
| D3 | | ERR$() | – return error string |
| D4 | | INSTR | – return string match position |

**New commands:**

| | | |
|---|---|---|
| D5 | ELSE | – part of IF. . . |
| D6 | RESUME | – restart after TRAP |
| D7 | TRAP | – detect error |
| D8 | TRON | – turn trace on |
| D9 | TROFF | – turn trace off |
| DA | SOUND | – output a sound |
| DB | VOL | – set sound level |
| DC | AUTO | – enable/disable auto line numbering |
| DD | PUDEF | – define PRINT USING symbols |
| DE | GRAPHIC | – set graphics mode |
| DF | PAINT | – fill area with color |
| E0 | CHAR | – display characters |
| E1 | BOX | – draw box |
| E2 | CIRCLE | – draw circle |
| E3 | GSHAPE | – display screen shape |
| E4 | SSHAPE | – save screen shape |
| E5 | DRAW | – draw dots and lines |
| E6 | LOCATE | – place pixel cursor |
| E7 | COLOR | – define screen color |
| E8 | SCNCLR | – clear screen |
| E9 | SCALE | – adjust graphics scaling |
| EA | HELP | – highlight error statement |
| EB | DO | – start a repeat block |
| EC | LOOP | – end a repeat block |
| ED | EXIT | – exit a repeat block |
| EE | DIRECTORY | – show disk directory |
| EF | DSAVE | – save to disk |
| F0 | DLOAD | – load from disk |
| F1 | HEADER | – format or clear a disk |
| F2 | SCRATCH | – remove file from disk |
| F3 | COLLECT | – disk block collect |
| F4 | COPY | – copy disk file |
| F5 | RENAME | – change disk file name |
| F6 | BACKUP | – dual disk backup |
| F7 | DELETE | – eliminate program lines |
| F8 | RENUMBER | – renumber program lines |
| F9 | KEY | – show or redefine function keys |
| FA | MONITOR | – go to machine language monitor |

**Language elements:**

| | | |
|---|---|---|
| FB | USING | – part of PRINT USING |
| FC | UNTIL | – part of LOOP |
| FD | WHILE | – part of D0 |

**New commands:**

| | | | |
|---|---|---|---|
| FE | 02 | BANK | – set memory bank |
| FE | 03 | FILTER | – define sound filter |
| FE | 04 | PLAY | – play musical sequence |
| FE | 05 | TEMPO | – define music speed |
| FE | 06 | MOVSPR | – position, move sprite |
| FE | 07 | SPRITE | – manipulate sprite data |
| FE | 08 | SPRCOLOR | – adjust sprite multicolors |
| FE | 09 | RREG | – assign sys registers to Basic variables |
| FE | 0A | ENVELOPE | – define instrument |
| FE | 0B | SLEEP | – pause for specified time |
| FE | 0C | CATALOG | – show directory |
| FE | 0D | DOPEN | – disk file open |
| FE | 0E | APPEND | – add to file |
| FE | 0F | DCLOSE | – disk file close |
| FE | 10 | BSAVE | – binary save |
| FE | 11 | BLOAD | – binary load |
| FE | 12 | RECORD | – position relative file |
| FE | 13 | CONCAT | – combine two data files |
| FE | 14 | DVERIFY | – disk verify |
| FE | 15 | DCLEAR | – clear all disk files |
| FE | 16 | SPRSAV | – store sprite string |
| FE | 17 | COLLISION | – sprite collision handler |
| FE | 18 | BEGIN | – start program block |
| FE | 19 | BEND | – end program block |
| FE | 1A | WINDOW | – define screen window |
| FE | 1B | BOOT | – load & run file |
| FE | 1C | WIDTH | – set graphic line width |
| FE | 1D | SPRDEF | – enter sprite definition mode |
| FE | 1E | QUIT | – not implemented |
| FE | 1F | STASH | – save to DRAM |
| FE | 20 | | not used |
| FE | 21 | FETCH | – get data from DRAM |
| FE | 22 | | not used |
| FE | 23 | SWAP | – exchange with DRAM |
| FE | 24 | OFF | – not implemented |
| FE | 25 | FAST | – run at 2mhz (80 col only) |
| FE | 26 | SLOW | – run at 1mhz |

# From Apple To Commodore And Back

Robert Adler
Montreal, Quebec

If you are like the many other computer owners who have mastered or at least de-mystified the BASIC which was included in your machine, then perhaps you would like to add a little more challenge to your BASIC programming.

If you are a Commodore owner, then BASIC 2.0 is what you are familiar with. In the past, you may have passed up good programs in a magazine or book that did not specialize in the computer which you use. You therefore probably missed out on some very interesting programs. No longer will you have to pass up those programs which were written for the computer which possesses the friendly name of the Apple.

The scope of this article does not include delving into complicated matters which may require special techniques, commands or Machine Language. Even so, we will accomplish a great deal with BASIC 2.0, better known as the BASIC of the Commodore 64, Vic-20, and PET computers.

The first BASIC which was ever written for a microcomputer was Altair BASIC. It was written by Microsoft founder, Bill Gates. It was actually the first piece of commercial software ever written for a personal computer. Out of that BASIC, which was later named Microsoft BASIC, grew other versions. Every company which put out a micro seemed to have its own version.

Two of these companies were Commodore and Apple. In 1977, Commodore introduced the PET 2001 computer. It had a tiny calculator type of keyboard, a nine-inch screen and a cassette drive all built into one unit. It had 8k of Random Access Memory (RAM), and a 16k BASIC in ROM. This was Commodore BASIC 1.0. The machine was later upgraded with an external cassette recorder, and provisions were made to the BASIC ROM to allow for connection of a disk drive. This was known as BASIC 2.0.

When Commodore introduced the 8032 business computer and the 4032 personal computer, they added commands to their original BASIC. These new commands allowed easier usage of their disk drives. This was BASIC 4.0. When Commodore tried to make the cheapest home computer they could possibly make, they introduced the world to the VIC-20. With the VIC-20, Commodore returned to BASIC 2.0.

In 1982, Commodore produced a computer which had almost thirteen times the amount of memory as the VIC-20, more advanced graphics and sound capabilities, but still the same BASIC 2.0 as was on their original PET computer. This was, of course, the Commodore 64.

Here we are today, left with almost the same BASIC as was used nearly 10 years ago. Large advances in microcomputers have happened since then. Apple computers started out with a very plain BASIC, called Integer BASIC. It was then upgraded to Applesoft BASIC. Applesoft had many new commands which made it an extended BASIC.

Over the years, thousands of programs were written using Applesoft BASIC. Many programs are still being written in this powerful version of BASIC. Because there are so many similarities between the two versions of BASIC, only the differences need to be discussed. For a complete listing of all of the keywords, consult the appropriate user's manual.

We will start off with a simple command in Applesoft called 'HOME'. This command is used to clear the screen and move the cursor to the top left corner of the screen. This is equivalent to the Commodore BASIC statement:

```
print"S"
```

The word HOME on the Apple may clear the screen but a HOME (lowercase reverse 's') on Commodore computers, does exactly what it says and no more; it puts the cursor in the home position.

The next keyword is just as easy. It is the Applesoft 'HTAB(x)' command where x is a number between 0 and 39. If you remove the H and add a semicolon to the end, making it TAB(x); you will have the equivalent in Commodore BASIC.

The next one is just a bit harder. It is the VTAB(x) command where x is a number between 0 and 23 to specify the screen line where the next printed line will go. This is replaced by executing a PRINT statement like the following:

```
print"sqqqqq";
```

The HOME character is followed by x number of CuRSoR down characters to produce the equivalent result. Please take into consideration that the Commodore 64 has 25 vertical lines and the Apple has 24.

There is another way to make the VTAB conversion. The second way is to use a subroutine such as this one:

```
4000 vt$ = "  " : d$ = "q"
4010 for cu = 1 to vt
4020 vt$ = vt$ + d$
4030 next
4040 print "s";vt$;
4050 vt = 0
4060 return
```

To use this routine, you simple set the variable VT to the number within the brackets of the VTAB command, and GOSUB 4000. The next line printed will appear on the proper vertically tabbed line. Please note that although your programs will be easier to read this way, the routine works considerably slower than the one liner discussed above.

*Note: A faster way to implement VTAB:*

```
4000 d$ = "sqqqqqqqqqqqqqqqqqqqqqqqq"
4010 print left$(d$,vt);
4020 return
```

Another easy conversion is the Applesoft INVERSE command. In Applesoft programs, all the text which is PRINTed to the screen after an INVERSE command, is reversed until the BASIC encounters a NORMAL command. In Commodore BASIC, INVERSE is replaced by:

```
print "r";
```

RVS is a special character achieved by simultaneously pressing the CTRL (pronounced Control) key and the numeric key marked 9 on the keyboard. To turn the reverse mode off, NORMAL is used in Applesoft while PRINT "<OFF>"; is used in BASIC 2.0. The word OFF refers to pressing the CTRL and zero (0) keys together.

There is one statement that you will find in Applesoft which looks the same but does not exactly act the same. To translate the Applesoft GET A$ (read: get 'A' string where 'A' can be any valid variable), you must not have any other statements on the same line except for the following translation:

```
10 get a$:if a$ = "" then 10
```

Of course the line number preceding the GET statement can be any line, but the same line number should be used after the keyword THEN. To get around having to always put this statement on its own line, and more closely simulate the Applesoft equivalent, use the following line instead:

```
poke 198,0:wait 198,1:get a$
```

This one is a lot better although it will only work on the Commodore 64 and VIC-20. The only thing that remains to be different still from the Applesoft GET A$, is the cursor that flashes while it waits for a keypress.

Using two POKEs, you can simulate a flashing cursor. Insert the two POKEs between the GET A$ and the IF-THEN statement as in this example:

```
10 get a$:poke 204,0:poke 207,0
20 if a$ = "" then 10
```

Possibly one of the easiest conversions would be the Applesoft CLEAR command which resets all variable pointers among other things. Take away the E and the A and you have the BASIC 2.0 command CLR.

Those are about all the commands that can be easily translated. There are other commands which are to follow in different categories that can not as easily be translated. The first category is graphic commands. The following list shows you what to look for before you try converting an Applesoft graphic program.

```
color = /hcolor =
draw/xdraw
gr/hgr/hgr2
plot/xplot
hlin/vlin
scale = /rot =
shload
scrn/pdl
```

The commands listed above are used for high and low resolution point plotting, line and shape table drawing. Commands that are similar can be used on the C64/VIC 20 with graphic command extension packages. The graphic screen on the Apple is 280 by 192 in the HGR2 mode while the high resolution screen on the C64 is 320 by 200. This similarity makes it easy to use high resolution parameters from Applesoft programs on the Commodore 64, once a graphic package is acquired either commercially or from the public domain.

The function PDL(x) where x is a number between 1 and 3 returns a number between 1 and 255 depending on the rotation of the paddle. To read the paddle on the Commodore 64 and get a result in the range of 0 to 255, use the following formula:

```
11 = peek(54297):p2 = peek(54298)
```

The variable P1 will show the results of paddle one in port one. P2 will show the results of paddle two in port one.

If you encounter the Applesoft PDL(x) functions, you might also find a series of peeks to test for a fire button. To test for a fire button on the Commodore 64, use the following formulae:

```
f1 = peek(56320) and 16 : f2 = peek(56321) and 16
```

The variable F1 will return a zero when the fire button on paddle one in port one is being pressed. F2 will return a zero when the fire button on paddle two in port one is being pressed. Each will return a four when no button is being depressed.

The next set of commands are the special editing and error trapping commands as shown in the following list.

```
trace/notrace
onerr/resume
del/pop
speed=/flash
```

The above commands can also be acquired by using an editing utility program, but are for the most part, not needed. The SPEED= and FLASH commands are keywords that just fancy things up a bit, and can easily be simulated in plain Commodore BASIC.

Let's take a short look at each one. The SPEED= command is usually used to slow down the speed of text output. At certain speeds, it can make text output resemble the speed at which 300 baud modems communicate. To implement a similar command on Commodore computers, we can use a very short subroutine. The subroutine shown here will expect the string variable TX$ to be equal to the text which you would like output in a slower than normal speed:

```
5000 for x=1 to len(tx$)
5010 print mid$(tx$,x,1);
5020 for t=1 to 333
5030 next t:next x
5040 return
```

After setting the TX$ variable to the text you want to print to the screen, all that is needed is a GOSUB 5000 statement. The output can be slowed down by increasing the delay loop in line 5020 and vice versa to speed it up.

To simulate the effects of the Applesoft command FLASH, which prints text in alternating reverse and normal characters, use the following subroutine:

```
6000 rv$=chr$(18):print
6010 print "<cursor up>";rv$;tx$
6020 if rv$=chr$(18) then rv$=chr$(146):goto 540
6030 if rv$=chr$(146) then rv$=chr$(18)
6040 for t=1 to 333:next
6050 get k$:if k$="" then 3010
6060 return
```

To use this subroutine, set TX$ to the text you would like flashed, and use the command GOSUB 6000.

The following set of commands deal with the internal workings of the computer or with the Input/Output (I/O).

```
himem/lomem
in#/pr#
store/recall
call
```

The above commands can be simulated on Commodores but will not maintain the same effect. HIMEM and LOMEM set high memory and low memory just like some pokes to locations in zero page such as 55–56 for setting the "highmem". IN# and PR# are similar to the INPUT# and OPEN statements except for the fact that a Commodore uses device numbers instead of slot numbers.

For example, to list a program to the printer on an Apple computer, you would type PR#1, assuming that the printer is in slot number 1. Control would then be transferred to the printer. Typing LIST would list the Apple program to the printer. To give control to the printer on a Commodore system, the following commands would have to be executed:

```
open 1,4:cmd 1:list
```

The one (1) may be substituted by any number from 1 to 255. A number higher than 127 sends an extra line feed after each carriage return. The four is the normal device number of the printer.

STORE and RECALL are used for writing files containing arrays to a cassette recorder. Storing files on disk or tape is not a hard task for a Commodore. It is however done differently. Explaining how to save sequential, relative and program files could fill up anywhere from a chapter to an entire book. For this reason, you should consult the proper manuals for each computer.

The CALL statement is exactly the same as the BASIC 2.0 SYS statement which calls up a Machine Language routine. If, however, you encounter a CALL statement in an Applesoft program, the program is using Machine Language which means that the conversion would consist of working with the Machine Language too. That is beyond the scope of this article.

Don't worry about those few commands that are not easily translated because just knowing the ones discussed here will be enough to translate hundreds of Applesoft programs. Revive an Applesoft program today!

# What is COMAL?

Michael J. Erskine
San Angelo, TX

COMAL stands for COMmon Algorithmic Language. It is a general purpose programming language conceived by two Danes in 1973, Borge Christenson and Benedict Lofstedt. It occurred to these gentlemen there existed a need for a high level, highly structured programming language to introduce non–structured thinkers to structured programming concepts.

Initially COMAL was a simple set of enhancements to BASIC, similar to BASIC 4.0. In the 13 years since its inception the language has evolved with the theory of structured programming. Today COMAL resembles BASIC in that COMAL retains some statements COMmon to many Algorithmic Languages; however COMAL is as different from BASIC as a Porsche is from a Model–T Ford. There was also a time when the only automobile one could own came in BASIC black and it was a very nice automobile. Given the exponential rate of growth of the hardware and software industries, is it really that hard to accept the fact that BASIC has become an antique? Is a Porsche a Model–T? Which would you most prefer to use for transportation?

COMAL is not BASIC, but learning COMAL is easier than learning BASIC, especially for a novice programmer. This is because the language was designed by educators for students of computer programming. Yes it is true that BASIC, among others, was designed under similar circumstances; but BASIC was designed before the surge toward structured programming. Giving BASIC and PASCAL due credit COMAL has retained the best features of both languages and has many new tricks of its own thrown in. We build upon what we already know and add to the store of knowledge through the creative process. This is true in any science and any art.

COMAL is easy to learn even though there are over 100 commands, statements, functions and procedures available in the Kernal definition. All these are machine independent. This means a program written using these Kernal commands will run on any computer running COMAL, just by typing it in! Remember the word, "COMmon"? COMAL is now available for the IBM PC series, in Europe. That's COMmon! Commodore 64's can also run COMAL in 2 versions, a disk loaded COMAL 0.14 and a cartridge COMAL 2.00, that's much more COMmon!

In addition to those 100 or so commands available in the Kernal, the programmer can build PROCedures and FUNCtions which effectively re–define the language. For instance, if you need a FUNCtion to figure the standard deviation of an array containing a set of test scores you can write such a FUNCtion and name it find'std'dev then call it using only its name. The operating system will jump to that FUNCtion and execute it (using the parameters you specify, if you wish) and then return, unless that PROCedure or FUNCtion makes subsequent calls. (more on COMAL names later) The cartridge version also allows calling EXTERNAL PROCedures and FUNCtions from disk, executing them, then continuing execution of the running program which called them. Try that in BASIC. GOSUB was not retained from BASIC, for obvious reasons.

For all you C–64 owners who realize the incredible, however often wasted, power of your VIC II and SID chips, the library of graphics and sound FUNCtions and PROCedures available will open a whole new world to you. There are 50 graphics "commands" such as GRAPHICSCREEN used to set hi–res or multi–color graphics and 49 others which control graphics and the TURTLE. "Yes, dear I'm playing with the TURTLE again. I can't help it, this LOGO EMULATOR is fascinating!". There are 32 sprite commands like IDENTIFY, DEFINE, SPRITEPOS (x,y) and the biggie ANIMATE. There are 19 sound commands allowing access to every possibility the SID chip can offer. They make programming a tune as easy as copying sheet music! There is a command for reading the joystick, and one for reading paddles. There are 6 light–pen commands and 7 special font commands which allow definition of a special font and placement of the font anywhere on any screen in any mode. I've a listing of a program about 3k long which plays music, uses 11 different sprites and draws with the TURTLE at the same time. The music is flawless and the little man walks across the screen exactly like a cartoon figure and the program contains NO MACHINE LANGUAGE. COMAL is very fast! It is so fast that I'd venture to suggest it may be possible for a clever programmer to write a procedure which makes the SID chip say "Hi, I'm SID and this is COMAL!" It may not be perfect but I'll bet it's understandable. Sorry, the sound, lightpen, joystick and paddle commands are only available in the cartridge version.

For the particular programmer COMAL offers 4 loop Fstructures:

(1) LOOP, EXIT, ENDLOOP
(2) FOR, ENDFOR
(3) REPEAT, UNTIL and
(4) WHILE, ENDWHILE.

There are two very powerful decision structures:

(1) IF, THEN, ELIF(else if), ELSE, ENDIF and
(2) CASE OF (variable), WHEN, OTHERWISE, ENDCASE.

The language also has built–in error handling routines which allow a programmer to TRAP an ERRor and REPORT it to the user via the ERRTEXT$ (which is defined by the programmer).

The interactive programming facilities are the equal of, perhaps better than, any language on any computer anywhere. You can PRINT AT (row,column), # USING or just plain PRINT. When you're not PRINTing you might INPUT AT (row,column-,number of characters) or place the CURSOR (row,column). If you are inputting data from the screen you will be pleased to find you are not able to leave the line or enter more data than specified in the number of characters. PAGE will clear the screen. KEY$ will check to see if a key was pressed and INKEY$ will wait until a key is pressed. If you PRINT SPC$ (8) eight spaces will be printed, but you can also PRINT TAB (8). TAB (8) won't print the spaces but will move the cursor. You can also set the ZONE 8 and use a comma outside of quotes to skip 8 spaces.

If you want to try your hand at writing a data base, you'll find relative file handling greatly simplified when you CREATE ("a relative file", number of records, record length). You might need to APPEND sequential files or DELETE any file also, or you may want to simply MERGE a couple of programs. COMAL provides easy to use facilities for working with up to eight disk units, dual or single. COMAL works with 1541 FLASH!(tm).

You say, "Well, that's all nice but what if I want to twiddle a bit or two?". Where shall I begin? Commodore's Assembler/Editor makes life much easier. After the code is written, just save it to disk and LINK it to your program, then you can SAVE the program and machine code to disk and they will both LOAD as a single module in subsequent LOADs. You can write several machine language routines and LINK them one at a time and they will not overwrite each other. You can twiddle individual bits with BITAND, BITOR, or BITXOR. COMAL can read and write binary, hexadecimal and ASCII files, and you can use any of the three types as constants in a program. It is possible to write machine language routines as PACKAGES (this is how graphics, sound, etc are included) and USE the package. There are people out there right now writing new packages of commands. After USE a package can be DISCARDed. You can USE more than one PACKAGE at once, subject to memory constraints.

If you do use up all 30K of work space you can inform your system that a PROC or FUNC is EXTERNAL and the operating system will LOAD and EXECute the routine called then return control to the main program carrying any changes or new data along. If that's not enough for your special menu–driven application, you can CHAIN a program from a running menu program and after it has been RUN for you, you can CHAIN back to the menu program.

The operating/programming environment is a real work of art. It includes what can only be called a programmer's word processor. The screen editor provides commands such as FIND "any string" and CHANGE "any string", "to any other string". There are 304 different error messages. Of those 30 are dynamic. This means they will return messages such as "count:unknown variable", "wrong type of:INPUT", "wrong type of:READ",etc. In other words the error message contains the name of the offending statement in many cases. The cursor is generally placed on the offending item or near it also. The error messages are non–destructive. After you have corrected the offending section the message will disappear and the over–written characters will be placed back on the screen!

The function keys are completely programmable using the DEFKEY function and they may programmed for use in direct mode and program mode. They can be easily reprogrammed from within a running program.

When in direct mode or while running a program you can use the 13 CTRL key functions, including such goodies as a true shades of grey graphics screen dump (CTRL D) and a text screen dump (CTRL P).

There is one other thing you should know about COMAL. There are some very serious programmers who are constantly writing and placing in the public domain some very sophisticated programs. COMAL really is the replacement for BASIC, LOGO and a few others. Take control of your C–64 get COMAL. In the opinion of anyone I've ever spoken with who has written in several languages and then tried COMAL, "COMAL does not have a future, COMAL *is* the future!"

### The Use Of Names in COMAL

I've been working in COMAL for about a year now. Happily, I never had a lot of experience with BASIC and therefore I am not having trouble with "BASIC thinking".

I don't presume to be a very good or experienced programmer but I have seen enough programs to express certain feelings about correct habits when programming in COMAL.

The idea behind COMAL is to be able to write programs which describe the solution to the specific problem being solved and reflect the logical procedures (steps) involved in that solution. In the words of Mr. Christensen, "It is a fact not to be overlooked that programming languages are not only used to control computing machinery, but also for COMMUNICATION OF IDEAS." This is a very powerful and wonderful concept.

COMAL allows us to use up to 78 characters in a variable, procedure or function name. If we are to communicate ideas we must use words. The more descriptive and specific our names the better the distant reader of our programs will understand them. This is critical to his or her ability to use the program. A COMAL program should be so descriptive when it is read that no further documentation is necessary! Program flow is documented by forced indentation (upon listing), calculations and most tests should be isolated and identified by the use of functions. Procedures should be used whenever a section of code is used more than once.

The names used to describe these procedures, functions and variables should be very descriptive. In a procedure which names all the colors by assigning a numeric value to a name for each color one should NOT assign variable names like bg: = 3 when he can say bluegreen: = 3. As a consequence of the above naming we would have two possible statements to change the PENCOLOR at some later time in the program, PENCOLOR(bg) and PENCOLOR(bluegreen). Which would you rather have to remember while you were writing the program? Which would you rather read if I had written the program?

In the same line of logic why should I call a procedure to figure the standard deviation of a set of test scores something like "std'dev(ts())" when I could call it with a statement like "figure'standard'deviation(test'scores())"?

The naming facilities available in COMAL are designed by the authors of the language to support the already excellent names of their statements and commands.

The effective COMAL programmer will carefully select the names in order to describe the PROCedure, FUNCtion or variable AND its use in the program.

He will also remember COMAL is NOT BASIC, not even enhanced BASIC. COMAL is COMAL !!! It's just better than anything else. Why try to describe a Porsche in terms of a Model–T?

## Cartridge COMAL 2.0 Library Descriptions

Library (page $80, $A59A-$BFF1):

| | | |
|---|---|---|
| A5C1 | | Sense routine |

PACKAGE **english:**

| | | |
|---|---|---|
| A686 | | Init routine |

PACKAGE **dansk:**

| | | |
|---|---|---|
| A68C | | Init routine |

PACKAGE **system:**

| | | |
|---|---|---|
| CA2F | | Init routine |
| A80B | PROC | setprinter(str) |
| A96A | PROC | hardcopy(str) |
| A976 | PROC | setrecorddelay(int) |
| A97D | PROC | setpage(int) |
| A984 | FUNC | inkey |
| A9B6 | FUNC | free |
| A9C3 | PROC | keywords'in'upper'case(int) |
| A9C6 | PROC | names'in'upper'case(int) |
| A9C9 | PROC | quote'mode(int) |
| A9E1 | FUNC | currow |
| A9E9 | FUNC | curcol |
| A9F6 | PROC | textcolors(int,int,int) |
| AA34 | PROC | defkey(int,str) |
| AA7F | PROC | showkeys |
| AB21 | PROC | bell(int) |
| AB2D | PROC | serial(int) |
| A7FF | PROC | settime(str) |
| A805 | FUNC | gettime |
| A878 | PROC | getscreen(REF str) |
| A87B | PROC | setscreen(REF str) |

Library (page $83, $800F-$C000):

| | | |
|---|---|---|
| 8081 | | Sense routine |

PACKAGE **graphics:**

| | | |
|---|---|---|
| 8CDC | | Init routine |
| 95CB | PROC | window(real,real,real,real) |
| 8F15 | PROC | viewport(int,int,int,int) |
| 8CA3 | PROC | drawto(real,real) |
| 8ADA | PROC | draw(real,real) |
| 8B06 | PROC | plot(real,real) |
| 8C7C | PROC | moveto(real,real) |
| 8AE8 | PROC | move(real,real) |
| A62A | PROC | circle(real,real,real) |
| A64F | PROC | arc(real,real,real,real,real) |
| A564 | PROC | arcl(real,real) |
| A55B | PROC | arcr(real,real) |
| 9426 | PROC | textstyle(int,int,int,int) |
| 9157 | PROC | plottext(real,real,str) |
| 8D9B | PROC | pencolor(int) |
| 8DBE | PROC | textcolor(int) |
| 8FC3 | FUNC | getcolor(real,real) |
| A37B | PROC | fill(real,real) |
| A380 | PROC | paint(real,real) |
| 9496 | PROC | background(int) |
| 9483 | PROC | textbackground(int) |
| 950B | PROC | border(int) |
| 951E | PROC | textborder(int) |
| 8E2A | PROC | graphicscreen(int) |
| 90FC | PROC | textscreen |
| A25D | PROC | splitscreen |
| A258 | PROC | fullscreen |
| 88FA | PROC | clearscreen |
| 895E | PROC | clear |
| A23B | PROC | showturtle |
| A248 | PROC | hideturtle |
| A20F | PROC | turtlesize(real) |
| 90A9 | FUNC | xcor |
| 90D6 | FUNC | ycor |
| 8CA3 | PROC | setxy(real,real) |
| 904D | PROC | setheading(real) |
| 9094 | FUNC | heading |
| 903F | PROC | left(real) |
| 903C | PROC | right(real) |
| 901A | PROC | forward(real) |
| 9017 | PROC | back(real) |
| 9536 | PROC | penup |
| 9542 | PROC | pendown |
| 954E | PROC | home |
| 9576 | PROC | wrap |
| 9584 | PROC | nowrap |
| A8D7 | FUNC | inq(int) |
| AFD7 | PROC | savescreen(str) |
| B027 | PROC | loadscreen(str) |
| ADF4 | PROC | printscreen(str,int) |

PACKAGE **turtle:**

| | | |
|---|---|---|
| 8CE2 | | Init routine |
| 9017 | PROC | bk(real) |
| 9496 | PROC | bg(int) |
| 88FA | PROC | cs |
| 901A | PROC | fd(real) |
| A248 | PROC | ht |
| 903F | PROC | lt(real) |
| 8D9B | PROC | pc(int) |
| 9542 | PROC | pd |
| 9536 | PROC | pu |
| 903C | PROC | rt(real) |
| 904D | PROC | seth(real) |
| A23B | PROC | st |
| 9483 | PROC | textbg(int) |
| 95CB | PROC | window(real,real,real,real) |
| 8F15 | PROC | viewport(int,int,int,int) |
| 8CA3 | PROC | drawto(real,real) |
| 8ADA | PROC | draw(real,real) |
| 8B06 | PROC | plot(real,real) |
| 8C7C | PROC | moveto(real,real) |
| 8AE8 | PROC | move(real,real) |
| A62A | PROC | circle(real,real,real) |
| A64F | PROC | arc(real,real,real,real,real) |
| A564 | PROC | arcl(real,real) |
| A55B | PROC | arcr(real,real) |
| 9426 | PROC | textstyle(int,int,int,int) |
| 9157 | PROC | plottext(real,real,str) |
| 8D9B | PROC | pencolor(int) |
| 8DBE | PROC | textcolor(int) |
| 8FC3 | FUNC | getcolor(real,real) |
| A37B | PROC | fill(real,real) |
| A380 | PROC | paint(real,real) |
| 9496 | PROC | background(int) |
| 9483 | PROC | textbackground(int) |
| 950B | PROC | border(int) |
| 951E | PROC | textborder(int) |
| 8E2A | PROC | graphicscreen(int) |
| 90FC | PROC | textscreen |
| A25D | PROC | splitscreen |
| A258 | PROC | fullscreen |
| 88FA | PROC | clearscreen |
| 895E | PROC | clear |
| A23B | PROC | showturtle |
| A248 | PROC | hideturtle |
| A20F | PROC | turtlesize(real) |
| 90A9 | FUNC | xcor |
| 90D6 | FUNC | ycor |
| 8CA3 | PROC | setxy(real,real) |
| 904D | PROC | setheading(real) |
| 9094 | FUNC | heading |
| 903F | PROC | left(real) |
| 903C | PROC | right(real) |
| 901A | PROC | forward(real) |
| 9017 | PROC | back(real) |
| 9536 | PROC | penup |
| 9542 | PROC | pendown |
| 954E | PROC | home |
| 9576 | PROC | wrap |
| 9584 | PROC | nowrap |
| A8D7 | FUNC | inq(int) |
| AFD7 | PROC | savescreen(str) |
| B027 | PROC | loadscreen(str) |
| ADF4 | PROC | printscreen(str,int) |

PACKAGE **sprites:**

| | | |
|---|---|---|
| 98B9 | | Init routine |
| 9979 | PROC | define(int,str) |
| 9B0D | PROC | identify(int,int) |
| 99AC | PROC | spritecolor(int,int) |
| 99BB | PROC | spritepos(int,int,int) |
| 9A4A | PROC | spritesize(int,int,int) |
| 9B46 | PROC | showsprite(int) |
| 9B52 | PROC | hidesprite(int) |
| 9A83 | PROC | spriteback(int,int) |
| 9A93 | FUNC | spritecollision(int,int) |
| 9A96 | FUNC | datacollision(int,int) |
| 9ABF | PROC | priority(int,int) |
| AB54 | PROC | linkshape(int) |
| AB5A | PROC | loadshape(int,str) |
| AB6E | PROC | saveshape(int,str) |
| 9B6F | PROC | movesprite(int,int,int,int,int) |
| 9A11 | PROC | stopsprite(int) |
| 9DFC | PROC | animate(int,str) |
| 9D13 | FUNC | moving(int) |
| 9D1F | PROC | startsprites |
| 9CEB | FUNC | spritex(int) |
| 9CFF | FUNC | spritey(int) |
| 9D3F | FUNC | spriteinq(int,int) |
| 9ECD | PROC | stampsprite(int) |

PACKAGE **font:**

| | | |
|---|---|---|
| CA2F | | Init routine |
| ABD0 | PROC | linkfont |
| ABDF | PROC | loadfont(str) |
| AC49 | PROC | keepfont |
| ABF1 | PROC | savefont(str) |
| AC57 | PROC | getcharacter(int,int,REF str) |
| AC87 | PROC | putcharacter(int,int,str) |

PACKAGE **sound:**

| | | |
|---|---|---|
| B287 | | Init routine |
| B2FE | PROC | note(int,str) |
| B3DE | PROC | pulse(int,int) |
| B3FA | PROC | gate(int,int) |
| B412 | PROC | soundtype(int,int) |
| B436 | PROC | ringmod(int,int) |
| B455 | PROC | sync(int,int) |
| B474 | PROC | adsr(int,int,int,int,int) |
| B4AD | PROC | filterfreq(int) |
| B4CD | PROC | resonance(int) |
| B4E6 | PROC | filter(int,int,int,int) |
| B508 | PROC | filtertype(int,int,int,int) |
| B52C | PROC | volume(int) |
| B543 | FUNC | env3 |
| B549 | FUNC | osc3 |
| B54F | FUNC | frequency(str) |
| B55B | PROC | setscore(int,REF int(),REF int(),REF int()) |
| B59F | PROC | playscore(int,int,int) |
| B5CD | PROC | stopplay(int,int,int) |
| B5FC | FUNC | waitscore(int,int,int) |
| B2E3 | PROC | setfrequency(int,real) |

PACKAGE **paddles:**

| | | |
|---|---|---|
| CA2F | | Init routine |
| B62C | PROC | paddle(int,REF real,REF real,REF real,REF real) |

PACKAGE **joysticks:**

| | | |
|---|---|---|
| CA2F | | Init routine |
| B6B9 | PROC | joystick(int,REF real,REF real) |

PACKAGE **lightpen:**

| | | |
|---|---|---|
| B77D | | Init routine |
| B7FA | PROC | offset(int,int) |
| B7D1 | FUNC | penon |
| B79B | PROC | readpen(REF real,REF real) |
| B820 | PROC | timeon(int) |
| B82A | PROC | delay(int) |
| B80D | PROC | accuracy(int,int) |

# COMAL for the Commodore 64

Chris Zamara, Technical Editor

## *An Introduction to COMAL: Better than BASIC*

*This article is not a product review, but presents information about a product which we feel is significant to the Commodore community.*

What is COMAL? If you're a COMAL fan and drive around with an 'I speak COMAL' bumper sticker, sorry for starting off with that question. But you see, COMAL isn't really all that well-known in North America yet, and many people just aren't sure. If you're one of the un-COMAL-ized, you may be delighted by what you read here. This article answers the *What* about COMAL and gives some programming examples just to give you a flavour of the language. A complete COMAL programming tutorial is beyond the scope of this article, but we hope to provide that kind of information in future articles.

COMAL (COMmon Algorithmic Language) is a programming language originally developed in Denmark by Borge Christensen, and is currently in widespread use throughout Europe. It is estimated that there are 100,000 COMAL users worldwide. The first version for Commodore machines ran on the PET/CBM, and was a public domain program, distributed in Canada by Commodore. The new C64 COMAL takes advantage of the 64's graphics and has been expanded from the original PET version. You can get the C-64 COMAL 0.14 system from the COMAL users group USA (see their address at the end of this article) or make a copy from someone who has it. You are encouraged to make copies of the COMAL system disk for friends or club members, as long as no profits are made and you copy the COMAL system disk unchanged.

COMAL has been described as a cross between BASIC and Pascal, with the good points of both languages and the drawbacks of neither. COMAL is as easy to use as BASIC, requiring little overhead to perform simple programs, but it has the speed, control structures and parameter-passing capabilities that BASIC lacks. It does have the powerful structures found in Pascal, but is not as restrictive to the programmer and is simple to use. As another bonus, it also contains the "turtle" graphics commands from LOGO. If this article so far sounds like an endorsement of the COMAL programming language, well so be it. Read on about the language's capabilities and you'll be able to judge for yourself.

There are two official versions of COMAL in widespread use right now. Version 0.14 runs from disk, and will leave your 64 with about 10K of free memory once the language is loaded into memory. (See the Article "Is 10K Enough?" elsewhere in this issue.) The disk version keeps all error messages on disk to save memory, so there is a slight delay before an error message appears. The newest version of COMAL, called 2.00, comes on a cartridge. The cartridge leaves about 30K of memory free for user programs, runs about twice as fast, and error messages are now fetched instantly. The cartridge also includes new features and commands not found in version 0.14. The points presented below will generally refer to both versions, with exclusive 2.00 features noted in the text.

COMAL is a cross between a compiler and interpreter, compiling each program line as it is entered. That means that you'll be able to edit and run your programs in the same kind of interactive environment that BASIC enjoys, but your programs will run about 5 to 10 times faster. It also means that the compiler looks at each program line right after you press RETURN, so you're informed of any syntax errors immediately. This prevents dumb errors from sneaking into an obscure part of a program that will only be executed, of course, when you're demonstrating it. If you enter a bad line, the computer beeps, gives a VERY descriptive message, and positions the cursor at the point the error occurred. Fixing the error or moving the cursor to another line will cause the error message to go away and leave the screen EXACTLY the way it was before, as if nothing had ever happened. This is good for the ego, since the computer is so willing to forget your errors and reward your successes.

### Programming in COMAL

Many of the actual keywords and functions in COMAL are the same as BASIC, so you won't be totally alienated the first time you fire it up. You still get PEEK, POKE, CHR$, INT, and a lot of other common functions. What makes COMAL better than BASIC is the structure of the language itself. The best thing is that you'll never need GOTOs again, and line numbers have no significance outside of editing — HOORAY! You don't have to worry about indenting your control structures properly, either; COMAL does it for you. The structures available are listed below:

```
IF(condition). . .THEN. . .ELSE. . .ENDIF
WHILE(condition). . .ENDWHILE
REPEAT. . .UNTIL(condition)
CASE(expression). . .WHEN(conditions). . .  OTHERWISE.
. .ENDCASE
FOR. . .ENDFOR       (like FOR..NEXT in BASIC)
TRAP. . .HANDLER. . .ENDTRAP  (error trap – only in
                                COMAL 2.00)
```

The above control structures are what gives COMAL a superior operative environment to BASIC. You never have to use confusing branches to transfer control to different sections of code, just use the control structures to create a conditional loop or perform a series of instructions or **procedures** based on a condition. Procedures (explained more later) are like super–powerful subroutines, and let you break a problem into simple, understandable modules. Any student of modern structured programming techniques will appreciate COMAL's set–up, and anyone used to Commodore BASIC will be amazed at how much simpler it is to program with an up–to–date, powerful language.

For. . .Next loops and assignment statements look different from BASIC, but if you enter them in BASIC form, COMAL will automatically convert for you! Version 2.00 will also show all keywords in uppercase when you list the program, and user–defined procedures, functions and variables in lowercase.

Besides the structures above, there are other major improvements that COMAL has over Commodore BASIC. For one, the use of long variable names, up to 78 characters long. And all characters are recognized, so 'ACCOUNTS_RECEIVABLE' and 'ACCOUNTS_RECEIVED' are two different variable names. (The underscore is a valid variable name character in version 2.00 and is selected with the back–arrow key.) The other important characteristic of COMAL is its use of procedures and functions.

### COMAL Procedures and Functions

When you define a procedure, it's like making your own COMAL keyword, since you call that procedure by just using its name, and passing as many parameters as that procedure needs. For example, a COMAL procedure to draw a square of a given size at a certain angle might look like this:

```
PROC square(size,angle)
  setheading(angle)
  FOR i: = 1 TO 4 DO
    forward(size)
    right(90)
  ENDFOR i
ENDPROC square
```

Now, to draw a square 25 units large at a 45 degree angle, you would just use the command:

```
EXEC square(25,45)
```

The EXEC statement is optional, so the statement could simply be:

```
square(var1,var2)
```

Want a nice design? No problem:

```
FOR n: = 1 to 50 DO
  square(n*4,n*5)
ENDFOR n
```

Once a procedure has been defined, you can use it from direct mode as well as program mode. A procedure definition can be placed anywhere in a program, and will not be executed unless called; it can't be 'fallen into' like BASIC subroutines. By building a program out of procedures, your code suddenly becomes simpler to understand and easier to de–bug. Furthermore, a procedure can be defined as 'CLOSED', meaning that all variables defined within the procedure are local. With a closed procedure, you can use any variable names you wish, such as 'I', without caring whether it's been used elsewhere. And in version 2.00, if you do wish to use a global variable within a procedure you can bring it in via the IMPORT command. And of course, you don't have to worry about what line numbers a procedure uses — it's always called by name. Procedures can be called from within other procedures, encouraging a "top down" programming technique, where a problem is broken into lower and lower levels of detail.

Since parameters are passed to a procedure as it is called, the problem of having to set up variables before calling a subroutine (like in BASIC) is eliminated. Entire arrays can be passed to a procedure, simply by including the array name in the parameter list. Procedures are used just the same way that built–in COMAL procedures are, making your subroutines into natural extensions of the language. In COMAL 2.00, Procedures can even be EXTERNAL, meaning that the procedure definition is on disk, and is brought in when the program calls it. This allows you to maintain a library of procedures on disk and use them from any program.

A few other notes about procedures. A procedure can be defined within another procedure, making it local (not executable from the main program or any other procedure). Another capability of procedures is that they can be used recursively, i.e. a procedure can call itself, using a new set of parameters each time it does. Using recursion often produces a very elegant solution to a seemingly difficult problem, for example drawing a binary tree or evaluating an expression.

Besides procedures, you can define your own functions in COMAL, which are used implicitly just like the BASIC functions SIN or LEFT$. For example, you may want a function to round any number to a given number of decimal places. Just define it like this:

```
FUNC round(number,places)
  mag: = 10 ↑ places
  RETURN INT(number*mag + .5)/mag
ENDFUNC round
```

Once this function definition has been included somewhere in your program (even at the end where it doesn't get executed), you can use it just as you would a built-in function, as in these examples:

```
amount: = round(cash,2)
PRINT " Time taken is approximately  ";round(
minutes/60,1); "  seconds. "
answer: = round(answer,precision)
```

Functions, like procedures, may also be declared as CLOSED, and can be used recursively.

### Features of C64 COMAL

Besides just the standard COMAL commands, version 0.14 and 2.00 have a whole array of commands to handle graphics and sprites. The cartridge version 2.00 is a complete implementation of COMAL–80, the current standard, but also contains extra commands in the way of *packages*, which can be invoked with the command:

```
USE packagename
```

The concept of packages works well, since the standard COMAL Kernel can be kept machine independent, and extra machine–dependent commands — such as those involving sound, graphics and sprites — can be added at will. That way, you only have to bring in what you need, and not use unnecessary processing time and memory. Some of the packages available with the cartridge version are FONT, GRAPHICS, JOYSTICK, LIGHTPEN, SOUND, SPRITES, SYSTEM and TURTLE. Each of these adds many powerful commands to the language, and additional packages can be loaded from disk. You can even create your own packages, customizing the language to your own needs; any package currently in USE will be saved along with your program.

Both COMAL versions contain "turtle" commands such as those found in the language LOGO. Turtle commands, combined with the procedure–oriented nature of COMAL, provide a very easy method to draw incredibly complex patterns on the screen. You simply move around a "turtle" (which appears as a triangle) by pointing him in the right direction and moving him a number of units forward or backward. The main turtle commands are: RIGHT and LEFT to turn the turtle a specified number of degrees; FORWARD and BACKWARD to move the turtle a specified number of units; PENUP and PENDOWN to tell the turtle whether or not to draw as it moves; PENCOLOR to select the drawing colour; and a host of other commands to show or hide the turtle, change his size, move him to an absolute position, find out his X and Y coordinates, fill in an area with a specified colour, and others. There is also a windowing capability to draw only within a pre–defined area or to scale the drawing area. The cartridge also contains some non–turtle graphics commands to draw arcs, circles, lines, and to retrieve information about current graphics and turtle settings.

If you're used to drawing patterns with packages like Simon's BASIC or other graphics utilities, turtle graphics are a real treat. Forget about calculating X,Y coordinates using number–crunching feats of math — just point the turtle in the direction you want and let him go. As an example, Listing 1 shows a COMAL procedure to draw an N–pointed star given its size and the number of points the star has. (It works well with anything but 6 points.) Note that the actual star–drawing takes place in only 4 lines, which just repeats the sequence FORWARD(size); RIGHT(angle) until all points are drawn. Try doing that with a cartesian–oriented graphics package! Furthermore, this procedure will draw the star wherever the turtle happens to be at the current time, so another procedure which was drawing something else could just call STAR wherever a star was needed in the picture. COMAL isn't just for drawing pictures, of course, but graphic examples show the flexibility of the language, and are certainly fun to write and run!

The COMAL cartridge includes commands to control sound, sprites, character fonts, joysticks, paddles, and a lightpen. But it is important to note that the COMAL system isn't just a different language for your C–64, it is an entirely new environment, replacing the 64's ROM set completely and turning the computer into a dedicated COMAL machine. The new environment is familiar, but contains features which help when editing. For one thing, the function keys are set up to generate oft–used commands such as LIST, RUN, TEXTSCREEN, SPLITSCREEN, FULLSCREEN, etc. (TEXTSCREEN and FULLSCREEN select either text or hi–res screen displays. The SPLITSCREEN command displays the hi–res graphics screen while setting a window of five text lines at the top of the screen. This text window can be positioned anywhere on the full text screen with the cursor up/down keys.) The function keys can also be re–defined as any string of text you wish. The cartridge provides a slew of other key–driven functions via control–key sequences. Pressing letter keys in conjunction with CTRL can give you a printer dump of the current text screen, move the cursor forward or back a word, erase to end of line, change border/screen and text colours, among other things.

The programming environment is further strengthened by the inclusion of FIND, CHANGE, AUTO, DEL, RENUM, and TRACE commands. The DEL command, used to delete a range of program lines, can also be used to delete an entire procedure or function by name. Incidentally, LIST works the same way. And the error messages are so descriptive and precise that it is possible to learn the syntax of the language simply by typing in random guesses and following the suggestions of the error messages, which say things like: ':=' or '(' expected, not integer constant. (If you wish, COMAL will even speak to you in Danish!) The overall programming environment is also enhanced by dozens of other clever touches like a pleasant bell sound when an error occurs, return from hi-res to text screen when a program is STOPped, word-wrap on program lines, and a smart INPUT statement which allows STOP key exits and glitch-free data entry.

Another unique feature of the language is its ability to process sound and sprite actions concurrently with program execution. You can set up any number of sprite operations which will be executed during the 60 cycle interrupts while the main COMAL program is running. There is also the MOVESPRITE command which simply tells the sprite where to move to and how fast, then continues program execution while the sprite does its thing. Likewise, music can be produced while a program is running by setting up a musical score in arrays and using the SETSCORE command. With its auto-animation capabilities, COMAL gives a simple way to implement normally complex operations.

COMAL's basic personality is a forgiving one, tolerating minor syntax aberrations and fixing them up when the program is listed. For example, to end a procedure, the ENDPROC command is used, followed by the procedure's name. If you leave off the procedure name, however, COMAL won't mind. The first time you RUN the program, it will figure out the correct name and put it in for you. The same goes for functions (ENDFUNC) and FOR. . .ENDFOR loops. So to an extent, COMAL documents your programs for you. Speaking of documenting, version 2.00 allows blank program lines to separate sections of code — just enter a line number by itself.

Another of COMAL's strengths is file handling and disk access. Programs can be stored and retrieved with LOAD and SAVE, or in sequential ascii format with ENTER and LIST. By opening a sequential file for input and using the SELECT command (in version 2.00), you can have BATCH files — that is, commands can be executed directly from a sequential disk file. Probably the best thing about COMAL's disk handling is the fact that random file access commands are built into the language, and COMAL fixes a bug that the 1541 has in dealing with random files.

COMAL has hundreds of features not found in BASIC, too many to list in this article. Things like a built-in string search command, no garbage collection delays, a PRINT USING command for formatted output, a ZONE command to set up tab fields, and dozens of little niceties that there isn't space to mention. At this point though, perhaps you have an idea of the scope and power of the COMAL system, and you can see why many who use it turn into big COMAL fans. Like the ones with the bumper stickers.

## COMAL Resources

There are quite a few books on COMAL, both texts and reference. There are also disks available from the COMAL users group packed with programs. The disks are under $10.00 each and there are over 2000 programs on 40 disks available by now. The COMAL users group USA publishes the magazine COMAL Today, which is filled with news, programs, and little tidbits about COMAL. A subscription to COMAL Today also gives you discounts on books and club disks. If you're interested in learning more about COMAL or wish to start using your COMAL system, a list of good references appear at the end of this article. Reviews of all of these books appeared in COMAL Today #7. These publications, the COMAL 0.14 system, or the cartridge are all available from The COMAL users group, USA. Several packages including COMAL, books and programs are also available. For more information, contact:

COMAL USERS GROUP, U.S.A., LIMITED
6041 Monona Drive
Madison, WI 53716

## COMAL Book List

"COMAL From A to Z"
Borge Christensen
– A reference of all COMAL commands; 64 pages

"COMAL Workbook"
Gordon Shigley
– An exercise text for beginners; 69 pages

"COMAL Library of Functions and Procedures"
Kevin Quiggle
– Reference guide for the included disk; 71 pages

"COMAL 2.0 Packages"
Jesse Knight
– How to add your own ML packages to COMAL; 108 pages

"Beginning COMAL"
Borge Christensen
– Informal introduction to COMAL by its creator; 333 pages

"Captain Comal's Graphics Primer"
Mindy Skelton
– COMAL graphics and sprites for beginners; 84 pages

"Cartridge Graphics and Sound"
Captain Comal's Friends
– Tutorial and reference for 2.0 extra package commands; 64 pages

"Commodore 64 graphics with COMAL"
Len Lindsay
– Complete organized reference for COMAL graphics commands; 170 pages

"Foundations in Computer Studies with COMAL"
John Kelly
– Programming textbook using COMAL; 363 pages

"Captain Comal Gets Organized"
Len Lindsay
– Writing a disk management system in COMAL, disk included; 102 pages

"Structured Programming with COMAL"
Roy Atherton
– How to write structured COMAL programs; 266 pages

"Cartridge Tutorial Binder"
Frank Bason & Leo Hojsholt–Poulson
– A tutorial specifically for the C64 COMAL 2.0 cartridge; 320 pages

"The COMAL handbook"
Len Lindsay
– **The** COMAL reference source, giving syntax and sample usage of all standard COMAL–80 commands

**Listing 1:** COMAL program to draw an N–pointed star –note how COMAL indents the control structures.

```
// "STAR" – this is a sample COMAL
// program to draw a star of any
// number of points.
// * transactor magazine 1985 –cz
USE graphics
USE turtle
splitscreen
PRINT ""147"",
size:=100
LOOP
  PRINT ""19"",
  INPUT "number of points? ": points
  clear
  xstart:=INT(160–size/2)
  ystart:=INT(100–size/2)
  moveto(xstart,ystart)
  pendown
  star(size,points)
ENDLOOP

PROC star(size,points)
  //** draw an N–pointed star **
  // first calculate the angle to
  // turn at each point
  CASE (points MOD 4) OF
  WHEN 0
    angvar:=points
  WHEN 2
    angvar:=points/2
  OTHERWISE
    angvar:=points*2
  ENDCASE
  angle:=180–360/angvar

  // now draw the star
  setheading((180–angle)/2)
  FOR i:=1 TO points DO
    forward(size)
    right(angle)
  ENDFOR i
ENDPROC star
```

# Is 10K Enough?

Steve Kortendick
Sun Prairie, Wisconsin

## Using The COMAL 0.14 System On The C64

Though available in many different formats, the most popular versions of COMAL are disk-loaded systems which reside in user memory. These releases of the language occupy space otherwise used by user programs. For example, a Commodore 64 running BASIC powers up with the message that there are about 38 kilobytes free, but when loaded with the COMAL system confesses to have only about 10k of free space remaining. This has been a source of consternation for those expecting 64k on their Commodores. But the real question is whether serious, sophisticated programs can be run in a small amount of user space like the 10k available with Commodore 64 COMAL.

I will admit at the outset that there are indeed some applications for which 10k is insufficient. It should come as little surprise, in fact, that there are applications for which the entire 64k of the Commodore 64 are to few, among them predicting the weather and flying a space shuttle. But within the domains for which we bought those machines, I have never found an instance in which I would prefer 38k of BASIC workspace over 10k of COMAL.

There's a certain elegance to doing a lot with a little. Countless hours of mainframe use, with seemingly limitless megabytes of "virtual memory", have not clouded the memories of coming home to my PET, powering up, and seeing

COMMODORE BASIC
7167 BYTES FREE

proudly displayed on the screen. At the time, this was the big 8k machine; they were still taking orders for the short-lived 4k model as well, with its "3071 BYTES FREE" message. Though I dreamed of the day I could add another 8k chip to that early home computer, it was a needless lust; seldom did the small memory size limit my activities with that machine.

With many of your programs (for some of you, all of your programs), the straightforward technique of simply storing your entire program and all necessary data simultaneously in the 10 free kilobytes will work quite well. Just compute merrily onward, and forget that some people with other applications might be having difficulty fitting everything into their machines. The remainder of this article is not for you.

First you can regain some free memory by "cleaning out" your program's *name table*. COMAL keeps every variable, procedure, and function name in a table. Once the name is in the table, it stays there, even if the variable isn't used any more. Misspelled names remain in the table as well, even if they are corrected in the program. COMAL saves the name table along with the program when you issue a SAVE command. Thus the old name table is reloaded with each LOAD. But, if you LIST the program to disk (LIST " NAME.L "), issue a NEW command, and then ENTER it back again (ENTER " NAME.L "), COMAL will rebuild the name table. You should have more free memory now.

Another very simple and efficient way of regaining lost space with COMAL is to hone down the size of your DIMs to what you actually need. In the DIMensioning of strings, COMAL reserves space in memory for the full number of bytes requested. Thus "DIM ADDRESS$ OF 1000" would reserve the full 1000 bytes of memory (plus some for the name and pointers) for the variable ADDRESS$, rendering that space unusable by any other variable. Recall that BASIC, in contrast, simply reserves a few bytes for the name (AD$ is all it can keep) and pointers, then claims additional speed as it is required. Though space is not wasted, the disadvantages with BASIC's technique are its speed (COMAL is over 79 times as fast in some string manipulations), its need for garbage collection (sometimes requiring several minutes to reclaim lost space), and its possibility of run-time errors ("OUT OF MEMORY ERROR IN 1230"). Likewise, when DIMensioning arrays ("DIM RANGE(-5:5, 1:25)"), use only the indices needed; more will rob you of potentially valuable space.

In BASIC, procedures (subroutines) are nameless creatures, identified only by their chance line number, and cannot receive parameters; functions are paltry one-line expressions identified by one letter and capable of handling only one true parameter and no decision logic. Both are consequently difficult to use and are avoided by legions of BASIC programmers. COMAL, in contrast, allows meaningful names to be assigned, parameters (even arrays) to be passed, and complex branching to be performed in both procedures and functions. This eliminates the need for the common variable reassignment necessary for most BASIC subroutines (eg. X1=L: X2=BR: T%=3: GOSUB 4250: IM=X4: REM SET UP VARIABLES AND INTERPOLATE). The use of procedures and functions not only eases the task of programming and debugging while making your code easier to read and understand, it also saves considerable space by not requiring you to repeat blocks of similar code. And the set-up required in BASIC is not needed in COMAL, simply call the procedure or function with the variable you need (eg. INTERMEDIATE := INTERPOLATE(LOW, HIGH, ACCURACY) ). And each procedure or function call takes only one byte, plus the parameters. Long variable names also take only one byte whenever used in a program, regardless of how long the name is. And the future is even brighter; the cartridge version of COMAL, in addition to freeing far more of the machine's memory, will allow external procedures to be called in from disk as needed and discarded from memory when they complete execution. *(The "future" is now here; the COMAL cartridge is available. See the "All about COMAL" article in this issue – T.Ed)*

Those of you who have been using COMAL for graphics applications are aware that there is no comparison with BASIC when considering the space required to use the 64's graphics abilities. BASIC needs confusing, tedious, and spacious strings of POKEs buried in FOR NEXT loops, while COMAL is content with simple keywords like FORWARD, LEFT, DRAWTO, and PLOT. Sprites, too, can be defined, moved, manipulated, and detected with clear COMAL statements such as HIDESPRITE, PRIORITY, SPRITEPOS, etc. Again, BASIC programmers are mired in a series of PEEKs and POKEs, ideally peppered generously with copious REMarks (and each COMAL keyword takes up only one byte each time used). Plus COMAL has reserved space for your graphics screens and sprite images right from the start. BASIC does not, forcing you to allocate it from within your program, losing about 4k. In addition, sound commands are available on the COMAL cartridge, but you can write your own sound procedures for the disk-based COMAL and easily create music and sound effects. The best that can be hoped for with BASIC is repeated code or a series of GOSUBs. The use of all these features can save considerable memory over an equivalent BASIC program.

Common structures in BASIC require a copious amount of space. The decision structure, for example in this menu option acceptance routine, is a series of:

```
IF(Q$="A"ORQ$="a"ORQ$="1")THEN
GOSUB1000:GOTO999
IF(Q$="C"ORQ$="c"ORQ$="2")THEN
GOSUB1200:GOTO999
IF(Q$="D"ORQ$="d"ORQ$="3")THEN
GOSUB1450:GOTO999
ER=3:GOSUB 2280
```

COMAL, however, allows a simple CASE statement:

```
CASE RESPONSE$ OF
  WHEN "A","a","1"
    ADD
  WHEN "C","c","2"
    CHANGE
  WHEN "D","d","3"
    DELETE
  OTHERWISE
    SIGNAL'ERROR(3)
ENDCASE
```

Besides being simple and non line-number oriented, COMAL is able to save the programmer significant amounts of space with such programming. In this example the difference is a savings of 59 bytes; BASIC would require 55% more space. Other structures which save bytes by eliminating hard-coded IF tests and subsequent complex branching are the ELIF and ELSE options of IF, together with WHILE and REPEAT UNTIL structures.

Other built-in features, if used properly, can also save bytes. The random number generator will provide you with integers within a specified range if you so desire, freeing you from the steps of multiplying by a range, adding one, and truncating (SHAKE:=RND(1,6) will assign the variable SHAKE with an integer between 1 and 6 inclusive). The ZONE command and PRINT USING will help you format a screen or printed page with far less character counting (and fuss) than the fixed zones found in BASIC. Another feature which saves space by eliminating a couple of IF THEN GOTOs on ST is the EOF system variable, which becomes TRUE (1) at the end of sequential files. Coupled with the UNTIL loop structure, it will save you not only space but also heartache. COMAL has other similar features which make programming not only compact but also quite straightforward. Further, such techniques are so clear that programs are easier to read without requiring nearly so much memory for REMarks – though *do not* neglect to comment (//) even your COMAL programs.

A technique I would recommend if you work with large amounts of data is to design your programs such that not all of the data are resident in the computer at any given time. A mailing list, for example, would not exist in an array in the machine, but would be on disk in a random access file. You might keep the index (key) values, or at least their sequence, in memory for faster access, however. Then you'd need only one name and address resident at any given time; updates can be done on an individual record basis. Another example might be statistical calculations on large sample populations. Thousands of values could be on the disk in a sequential file, and you might read through them, summing samples, squares, cross-products, etc., retaining only those sums in memory. After a pass or two through the file, you'd have everything you need for all kinds of statistical calculations, yet very little need be kept in memory at once.

The time may come, despite all of the above-mentioned techniques, that you'll find yourself hemmed in by the 10k limit imposed by the disk-loaded version of COMAL. Are you doomed to return to programming in BASIC? Not at all. Your program and data size can be up to whatever you have available on disk(s), at least 170k. This is accomplished through a memory management technique known as overlays. All that is required is that the currently executing program prepare any data necessary for the next program, then CHAIN the new program into the computer. This eliminates the program that did the CHAINing, and passes control of the system to the beginning of the new program. For example, a program called COMPUTE'MEANS could finish its task, and end up with a statement CHAIN " DO.DELTA.SQ " which would effectively LOAD the program DO.DELTA.SQ from disk and begin its execution.

This CHAINing technique is particularly easy to implement in a menu-driven system with clearly distinguishable subtasks. THe menu programs need only display a menu on the screen and ask for a response through a GET or INPUT statement. The rest of the program might then say

```
REPEAT
  CASE RESPONSE$ OF
    WHEN "I"
      CHAIN "INPUT'ROUTINE"
    WHEN "F"
      CHAIN "FIX'DATA'ROUTINE"
    WHEN "C"
      CHAIN "SCRATCH'FILE"
    OTHERWISE
      INPUT "Enter I, F, C or S:": RESPONSE$
  ENDCASE
UNTIL RESPONSE$ IN "IFCS"
```

Each CHAINed program would end with CHAIN " MASTER-'MENU "

There is a potential problem with this chaining technique: it resets all user variables and DIM statements. At times this makes communication between CHAINed programs somewhat difficult. Three techniques are fairly easy to use.

The first is simply to find some unused bytes in a safe place in memory (the home of an unused sprite is often handy) and POKE the values necessary into this sequence of bytes. This is quick and easy for small amounts of data, does not change the screen, and causes no I/O delays.

The second technique is to use the screen. You can either POKE to the screen as above, or you can PRINT to the screen, using cursor controls for positioning if needed. If you don't want the information seen, simply make your pencolor the same as the background color; the information will be there, but will be hidden. The alternative, of course, is to make the information seen, making sure you put things where they'll look good. Here, getting the data back can be quite interesting. Of course you still have the alternative of PEEKing at what you want, but there's a far more enjoyable way. You can OPEN the screen (device 3) as an input file, then INPUT directly from the screen after positioning the cursor. This input from the screen technique is explained in the COMAL HANDBOOK, first edition, page 204 (UNIT) and 123 (OPEN), and in the first issue of COMAL TODAY newsletter. What happens is that COMAL treats the screen as a sequential file, with each line seen as a record. You merely INPUT FILE from the screen, getting any information you need.

A third technique for passing data between CHAINed programs is to use intermediate storage. The CHAINing program could OPEN a disk file, WRITE its parameters to that file, CLOSE the file, then CHAIN the next program. The CHAINed program, for its part, would DIMension whatever were necessary, OPEN the parameter file, READ the parameters, CLOSE the file, and perhaps even scratch (DELETE) it. Then it would get down to business as usual. This method has the least of the kludge in it, but requires some time-consuming I/O. As always, there's a trade-off.

As I admitted in the beginning of this article, there are applications for which 10k of user memory will be insufficient. But several techniques have been presented which should help you pare down the size of your programs, and, if necessary, overlay them with others. Though there is some cost involved in the careful planning and space-conscious programming of a COMAL program, I find it far more pleasant and far less time-consuming than programming in BASIC, despite the latter's 38k available.

# GO LOGO GO

Howard Strasberg
Don Mills, Ontario

Tried Logo? No? Break the ice with this.

NOTE: Although this article is written primarily for the Commodore 64, Logo is very similar on most machines. Therefore many of the things that are mentioned can also be used on other computers.

Logo is a language that should scare no one. It really is quite easy to use. It has a reputation for being so simple yet powerful, that even very young children can draw interesting designs. Logo is a great tool for graphics as compared to BASIC. Logo allows fast and easy use of the hi-resolution screen. If you have ever tried bit-mapping in BASIC, you will know what I mean. It is a pain and it is slow. Try machine language and spend years typing it in! Logo is the perfect solution! Logo is also quite a bit friendlier than BASIC. If you do something wrong in BASIC, the computer responds with a ?SYNTAX ERROR. I find that very rude. Logo is different. In Logo, when you either accidentally or purposely make a mistake, you get a THERE IS NO PROCEDURE NAMED . . . .

When you understand Logo, it is quite friendly. You see, Logo uses what it calls procedures to do anything. A procedure which comes with Logo, something that is already programmed, is called a Logo Primitive. Something that you make, let us say a program to draw a square, is called a Procedure. And to RUN a Procedure in Logo, all you have to do is enter the name of it. So, if you had a procedure to draw a square, and called it SQUARE, then a square would be drawn by typing SQUARE. And if you typed SQURE (instead of SQUARE), then Logo would respond: THERE IS NO PROCEDURE NAMED SQURE. I'll talk more about procedures later in this article.

Let us begin. As soon as you have loaded LOGO, type DRAW. This tells Logo that you wish to have a fresh hi-res screen to draw on. The screen will clear, there will be a cursor flashing on the lower part of the screen and there will be a triangle in the middle. This triangle is what we call the turtle. The turtle does all of our drawing for us.

We want to move the turtle up. Only in Logo there is no such thing. Instead, we use FORWARD. The command FORWARD moves the turtle in the direction the turtle is pointing. It is very important that you understand FORWARD and the difference between it and "going up". Now, we cannot just say FORWARD. We need to say how many pixels forward. Type FORWARD 100. The turtle now should have moved 100 pixels forward. The opposite function of FORWARD is BACK. Type BACK 100. The turtle should now be in its home position (center of screen). Another way of returning the turtle home is the command HOME (Logo is so easy to grasp).

Now, if we are going to draw anything that looks half decent, we must be able to move more than forward or back. Type FORWARD 100. Now, we want to move 100 pixels to the right. There is a command RIGHT. However, it does not move the turtle right, it turns the turtle right. So, type RIGHT 90. This turns the turtle right 90 degrees. You must understand that RIGHT 90 rotates the turtle 90 degrees FROM THE DIRECTION IT IS FACING. If the turtle is facing south, then RIGHT 90 will make it face west. To actually SET the turtle's HEADING to 90 degrees (face east), type SETHEADING 90. Now that we have it facing right, we can say FORWARD 100. Type RIGHT 90 again and FORWARD 100 again. Try to complete the square.

We can also have the square on the other side, left of the middle of the screen. To do this, substitute the RIGHT with LEFT. Carry out the following commands:

```
FORWARD 100
LEFT 90
FORWARD 100
LEFT 90
FORWARD 100
LEFT 90
FORWARD 100
LEFT 90
```

Logo, being the powerful language that it is, can do this with much less typing and much faster. It is kind of like a FOR..NEXT loop in BASIC. We use the REPEAT command. The format is:

```
REPEAT xx (procedure)
```

Where xx contains how many times to repeat whatever is inside the square brackets. Type DRAW. Now, use REPEAT to draw our LEFT square:

```
REPEAT 4 (FORWARD 100 LEFT 90)
```

Experiment now, making different sized squares, rectangles, triangles and, for a challenge, circles.

There are some Logo commands which determine the specifics of the pen (the instrument the turtle uses to draw). They are also straight forward. If you want to move the turtle somewhere, but not leave a line while it is going there, just enter PENUP. Penup is like a printer with no ribbon pressing on the paper. The turtle (pen) will move where you want without making a line. To continue drawing, give the PENDOWN command. PENERASE can only erase a line with the PENDOWN. This is the turtle's ability to move somewhere and erase anything it happens to go over. To do this, enter PENERASE. To return to normal from this one, we must change the turtle's colour back to 1 with PENCOLOR 1. As a matter of fact, Logo's turtle can draw in 16 different colours, numbered from 0–15. The following is a chart of the number and its corresponding colour:

| | | | |
|---|---|---|---|
| 0 | black | 8 | orange |
| 1 | white | 9 | brown |
| 2 | red | 10 | lt.red |
| 3 | cyan | 11 | grey 1 |
| 4 | purple | 12 | grey 2 |
| 5 | green | 13 | lt.green |
| 6 | blue | 14 | lt.blue |
| 7 | yellow | 15 | grey 3 |

Again to access these colours, type PENCOLOR x, where x is the numerical value of the colour you wish. The colour of the background where the turtle lives can be changed with BACKGROUND x.

To get a better understanding of the PEN functions, enter the following commands:

```
DRAW FORWARD 100
PENERASE BACK 100 PENCOLOR 1
LEFT 90 PENUP FORWARD 50
PENDOWN HOME
```

Press F1. You now see all of the information you have entered in the last few minutes. This is known as TEXTSCREEN, and can also be accessed by that name. Experiment with F3–SPLITSCREEN and F5–FULLSCREEN.

Before talking about the procedure topic which I touched on earlier, I would like to bring your attention to short forms. Most primitives in Logo do have an abbreviation. If the name of the command is a compound word, then the short form is the first letter of each of the two words (The short form for PENCOLOR is PC). If it is not a compound word, then the abbreviation is the first and last letter (The short form of FORWARD is FD). In some cases, no short form exists, in which case you must type in the whole word (I know what you are thinking – NOW he tells me about short forms!!!) RT 90 is identical to RIGHT 90.

Now, about procedures. Let's make a procedure that draws a square. We will brilliantly call it SQUARE. Type:

```
TO SQUARE
```

The screen will clear. (MISC NOTE: The editing system in Logo is much different from that of BASIC. I do not intend to go into the details of this editor. Try not to make a mistake. To find out more about the editor, consult a reference book, have someone teach you, or just experiment. Experimentation is the method I used.) You are now ready to define a procedure. This procedure will be quite brief. We'll make our square slightly smaller (80 instead of 100). Type:

```
REPEAT 4 (FD 80 LT 90)
```

That is it! Press CTRL–C and the procedure will be defined. Now type DRAW. You will see the turtle. Type SQUARE. Voila! I believe it is time for a design. Type:

```
REPEAT 36 (SQUARE RT 10)
```

This draws 36 squares, each 10 degrees apart. As you can see, Logo is doing quite a lot of things, and quite easily too. Remember earlier I challenged you to draw a circle? Here is how. All you do is create a 360–sided figure and have the turtle rotate 1 degree in between sides:

```
DRAW
REPEAT 360 (FD 1 RT 1)
```

Logo also can STAMP a CHARacter on the screen, in case you want your design to say something. Type:

```
DRAW STAMPCHAR "L
```

and an L will be placed behind the turtle. However, in order to get a clear view of our STAMPed CHARacter, we must HIDE the TURTLE, which brings me to my next point. If at anytime you want to draw without showing the turtle, simply type HIDETURTLE, or HT. To bring it back to life, enter SHOWTURTLE, or ST.

As you have undoubtedly noticed, Logo can accomplish a lot. And everything it does is done logically and powerfully. Many interesting and colourful shapes and designs can be drawn. However, Logo is capable of doing much more than just drawing. Logo can play music, do mathematics, handle sprites, do amazing things with words, and much more. If you find Logo interesting now, keep at it. You will find it demands your attention, but also offers entertainment and excitement. Good luck. . .

**Editor's Note:** *I believe COMAL contains more LOGO type commands than LOGO itself. If you want to try your hand at LOGO, then COMAL is a good place to start.*

# Hidden Op–Codes

Jim McLaughlin
Ottawa, Ontario

*. . .For the record, all of the commands talked about in this article behaved identically on my 6502 and on my 6510. . .*

All computer users have experienced the problem of their machine crashing due to the microprocessor's failure to understand certain commands.

In this article I will attempt to clarify what happens at the machine level when a member of the MCS6500 microprocessor (CPU) family encounters an unrecognizable command.

### Some Microprocessor History

MOS Technology, one of the companies that manufactures the MCS6500 family of microprocessors, claims that all of their chips can execute 146 instructions, in 13 addressing modes. In fact, the Commodore 64 Programmer's Reference Guide notes: "COMMODORE SEMICONDUCTOR GROUP cannot assume liability for the use of undefined OP CODES". Each instruction is identified by an eight bit number, and if my math is correct, that allows for 256 possibilities. What happens if the CPU is requested to execute one of the remaining 110 codes, you ask? Any number of things can happen, ranging from a "no operation", to a "crash".

A look at a table of documented instructions will show that there are no op–codes in the ranges x3, x7, xB, and xF. where 'x' is any hexadecimal digit. Right away, 64 of the 110 instructions are found. Another curious fact is that there is only one instruction in the x2 range. Again, another 15 instructions are accounted for. Most, but not all of the tables will also list the command ROR and its 5 addressing modes. Since this command was omitted in 6502's built before 1977, software written for the 6502 must account for the missing instruction. This leaves 26 unrelated instructions spread throughout the rest of the ranges.

Back to the make–up of the op–codes for a moment. The eight bits (76543210) that make up the op–code are arranged in the following manner. Bits 2, 3, and 4 are used to calculate the addressing mode (see table 1). Bits 0 and 1, according to the first two bits rule, are used to determine the type of instruction, and surprisingly enough, are never both set to 1 in any of the documented op–codes. Apparently when these bits are set at 11, the instructions for 10, and 01 are executed one after the other, usually in that order. Generally this is the case for the x3, x7, xB, and xF commands.

**TABLE 1:** Addressing Modes – Op-code = xxbbbxx

| first b | last two bb |
|---|---|
| 0 not post–indexed | 00 (Ind,X)<br>01 Zero Page<br>10 Immediate/ Accumulator |
| 1 post–indexed | 00 (Ind),Y<br>01 Zero Page,X<br>10 Absolute,Y<br>11 Absolute,X |

In the new list of commands, one will find that there are 6 new NOP's. Each takes up the same number of bytes and the same time to execute as the original NOP. There are also "skip a byte" and "skip a word" (a two byte number). The SKB command takes up 2 bytes and the execution times range between 2 and 4 clock cycles. The SKW command takes up 3 bytes and the execution time is 4 clock cycles.

If you expected that the times for execution of these commands would be the sum of times of the individual commands, you would be wrong most of the time. It turns out that most of the time used by the CPU to complete an instruction is taken up by its addressing of the data used. Hence the time for the two instructions is not much more than that of one of the instructions. (See table 2 for a list of op–codes, addressing modes and timing values).

I found the CPU's execution times by employing a simple routine that carried out the command about 14 million times. This was compared to the time taken to execute a command with a known number of clock cycles. Having set up standard times, I was able to predict the timing of any, new or old. Later testing showed that the loop could have been executed only 100,000 times and the commands would have still been predictable.

One of the advantages to using these "new" commands is the saving of much time and space. For example, if you wanted to load the accumulator and the X–register with the same data, such as in absolute addressing mode, in normal assembly language it would be written something like this:

```
ad 01 08    lda $0801
aa          tax
```

This short routine takes up 4 bytes and takes 6 clock cycles to complete. However, if the same routine was written with the "undefined op-codes", it would be written as follows,

```
AF 01 08    lax $0801
```

The number of bytes consumed is 3, and the execution time is 4 cycles, a saving of 1 byte and 2 cycles. In a substantial loop the saving of the 2 cycles might cut execution time by one third.

Another value of the op-codes is that at this point, there are no disassemblers that can handle them. This makes it very easy to protect software since a pirate cannot make any sense of the code even if he can view it in memory.

For example, consider the following program:

```
AF 01 01    lax $0101
0C 00 00    skw
6F 16 01    rra $0116
```

If this routine was disassembled with a normal disassembler, it would result in:

```
AF          ???
01 01       ora ($01,x)
0C          ???
00          brk
00          brk
6F          ???
16          asl $01,x
```

This would make a very strange looking program, but would run without any trouble.

The problem of incompatiblility should also be considered. The main reason why the new commands are not documented by the manufacturer is that they may not be present in all chips. Even if there is a command with the same number, they may not execute in exactly the same way. When a new chip such as the 6510 was introduced into the market, the whole internal structure was changed. Consequently some of the new commands did not work in some situations. For the record, all of the commands talked about in this article behaved identically on my 6502 and on my 6510.

Some of the commands are so specialized that they are only used in very rare circumstances.

As previously mentioned, there are 15 commands in the x2 range that are not officially documented. I have given 12 of these commands the name "crash immediately", after the Z-80's command "halt and catch fire" or "crash and burn". The command CIM causes the chip to loop forever, or until halted. The only explanation that has been brought forward is that all the branch commands end with a 0 and that the x2 commands are "near neighbours".

The second last group of undocumented op-codes lies in the group of individual commands. In other words, there is only one addressing mode for each of these commands. These commands include ALR, ARR, MKA, MKX, OAL, and SAX. For a complete description of these, see table 2.

The last few undocumented op-codes lie in the group I like to call "the unknown" or the "peculiar". These are four commands that do not seem to perform the same way on two different CPU's. The four bytes are 89, 9C, BB, and EB.

The 89 byte looks as if it should be STA Immediate, but that is impossible. It does, in fact take up 2 bytes and 2 clock cycles.

The BB byte looks as if it might be OAL ABS,y, but it is not and the only thing that can be said about it is that it takes up 3 bytes, and I was never able to find out how many cycles it took.

The second last byte is 9B. This one is very strange, in that it is the missing STA command. It now gives the programmer the ability to store the accumulator to an absolute address, indexed by the X-register.

The last peculiar byte to be accounted for is EB. Not much can be said about this command either, other than it takes up 2 bytes of memory, and 2 clock cycles to execute. A little testing has shown that the EB byte seems to act just like the command AND,zero page. In side by side testing the two provided the same answers. It is interesting to note that the original AND takes 3 clock cycles, as opposed to the the new one which only took 2.

I have taken great pains to make sure that all that is written here is correct. However, the commands may work differently on other machines. If you want to write any programs using the new op-codes, I suggest that it be tested on several machines before assuming that it is correct. Most of the commands appear to be nearly universal in all MCS6500 family CPU's, especially the ones in the x3, x7, xB, xF ranges. Remember, if at first the new commands don't work, there is always the documented commands on which to fall back.

**Sources Consulted**

1. Extra Instructions, Joel C. Shepherd, Compute!, Oct. 1983.
2. Programming the PET/CBM, Raeto Collin West.

## Table 2: Commands, Modes And Timing Values

**Legend:**

| | | | |
|---|---|---|---|
| A - accumulator | C - carry flag | & - logical AND | → - transfer to |
| M - memory location | + - add | V - logical OR | $xx - zero page addressing |
| X,Y - registers | − - subtract | ⊻ - logical EOR | * - add one cycle if crossing boundary |
| | | | $xxxx - absolute addressing |

| Op-Code | Operation | Addressing Mode | Hex Code | Clock Cycles | Flags Affected |
|---|---|---|---|---|---|
| **ALR** | LSR (A & M) → A | Immediate | 4B | 2 | NZC |
| **ARR** | ROR (A & M) → A | Immediate | 6B | 2 | NZC |
| **ASO** | (ASL M) ⊻ A → A | Absolute | 0F | 6 | NZC |
| | | Absolute,X | 1F | 7* | NZC |
| | | Absolute,Y | 1B | 7* | NZC |
| | | Zero page | 07 | 5 | NZC |
| | | Zero page,X | 17 | 6 | NZC |
| | | (Ind,X) | 03 | 8 | NZC |
| | | (Ind),Y | 13 | 8 | NZC |
| | | Immediate | 0B | 2 | NZC |
| **AXS** | A & X → A | Absolute | 8F | 4 | NC |
| | | Zero page | 87 | 3 | NC |
| | | Zero page,Y | 97 | 4 | NC |
| | | (Ind,X) | 83 | 6 | NC |
| | | (Ind),Y | 93 | 6 | NC |
| **DCM** | A − (DEC M) | Absolute | CF | 6 | NZC |
| | | Absolute,X | DF | 7* | NZC |
| | | Absolute,Y | DB | 7* | NZC |
| | | Zero page | C7 | 5 | NZC |
| | | Zero page,X | D7 | 6 | NZC |
| | | (Ind,X) | C3 | 8 | NZC |
| | | (Ind),Y | D3 | 8 | NZC |
| **INS** | A−(INC M)−C → A,C | Absolute | EF | 6 | NZCV |
| | | Absolute,X | FF | 7* | NZCV |
| | | Absolute,Y | FB | 7* | NZCV |
| | | Zero page | E7 | 5 | NZCV |
| | | Zero page,X | F7 | 6 | NZCV |
| | | (Ind,X) | E3 | 8 | NZCV |
| | | (Ind),Y | F3 | 8 | NZCV |
| **LAX** | M → A,M → X | Absolute | AF | 4 | NZ |
| | | Absolute,Y | BF | 4 | NZ |
| | | Zero page | A7 | 3 | NZ |
| | | Zero page,X | B7 | 4 | NZ |
| | | (Ind,X) | A3 | 6 | NZ |
| | | (Ind),Y | B3 | 5 | NZ |
| **LSE** | (LSR M) ⊻ A → A | Absolute | 4F | 6 | NZC |
| | | Absolute,X | 5F | 7* | NZC |
| | | Absolute,Y | 5B | 7* | NZC |
| | | Zero page | 47 | 5 | NZC |
| | | Zero page,X | 57 | 6 | NZC |
| | | (Ind,X) | 43 | 8 | NZC |
| | | (Ind),Y | 53 | 8 | NZC |
| **MKA** | A & #$04 → A | Absolute | 9F | 5 | NZ |
| **MKX** | X & #$04 → A | Absolute | 9E | 5 | NZ |
| **OAL** | (AV#$EE)&M) → A,X | Immediate | AB | 2 | NZ |
| **RLA** | (ROL M) & A → A | Absolute | 2F | 6 | NZC |
| | | Absolute,X | 3F | 7* | NZC |
| | | Absolute,Y | 3B | 7* | NZC |
| | | Zero page | 27 | 5 | NZC |
| | | Zero page,X | 37 | 6 | NZC |
| | | (Ind,X) | 23 | 8 | NZC |
| | | (Ind),Y | 33 | 8 | NZC |
| | | Immediate | 2B | 2 | NZC |
| **RRA** | (ROR M) + A + C → A,C | Absolute | 6F | 6 | NZCV |
| | | Absolute,X | 7F | 7* | NZCV |
| | | Absolute,Y | 7B | 7* | NZCV |
| | | Zero page | 67 | 5 | NZCV |
| | | Zero page,X | 77 | 6 | NZCV |
| | | (Ind,X) | 63 | 8 | NZCV |
| | | (Ind),Y | 73 | 8 | NZCV |
| **SAX** | (A&X)−M−C → X | Immediate | CB | 2 | NZCV |
| **XAA** | (X & M) → A | Absolute,Y | 9B | 5 | NZ |
| | | Immediate | 8B | 2 | NZ |

There are also four implied commands.

| Command | Hex Code | Clock Cycles | Command | Hex Code | Clock Cycles |
|---|---|---|---|---|---|
| **NOP** | 1A | 2 | **SKW** | 0C | 4 |
| | 3A | 2 | | 1C | 4 |
| | 5A | 2 | | 3C | 4 |
| | 7A | 2 | | 5C | 4 |
| | DA | 2 | | 7C | 4 |
| | FA | 2 | | DC | 4 |
| | | | | FC | 4 |
| **SKB** | 80 | 2 | | | |
| | 82 | 2 | **CIM** | 02 | – |
| | C2 | 2 | | 12 | – |
| | E2 | 2 | | 22 | – |
| | 04 | 3 | | 32 | – |
| | 14 | 4 | | 42 | – |
| | 34 | 4 | | 52 | – |
| | 44 | 3 | | 62 | – |
| | 54 | 4 | | 72 | – |
| | 64 | 3 | | 92 | – |
| | 74 | 4 | | B2 | – |
| | D4 | 4 | | D2 | – |
| | F4 | 4 | | F2 | – |

# A Comparison Of CPUs: The MOS 6502, Motorola 6809, and Motorola 68000

Richard Evers, Editor

To enlighten your day, our chip comparison will be slightly delayed in order that we may bring you a quick chip history lesson as it applies to the world of Commodore. Our story begins before MOS technology was formed, with the hero of our tale being a very talented individual by the name of Chuck Peddle. Back in the days of old, the name Peddle was synonymous with Motorola. In particular, it was Chuck Peddle who played a key role in the design of Motorola's first eight bit processor, the 6800. As history advanced, Chuck Peddles knack of leading the way in technological break throughs seemed to become his trademark.

As time progressed, the 6800's evolution continued due to the efforts of many people at Motorola until the 6809 chip, a pseudo 16 bit delight with an 8 bit data bus, was conceived. The chip was an instant, limited success for Motorola. Great chip, kind of costly to make. A mini interjection: A joint venture between The University of Waterloo and BMB Compuscience back in the early 80's produced what became later known as the SuperPET Microcomputer. The system was based on the Commodore 8032 microcomputer, but was further refined to include a Motorola 6809 processor, 64k of extra RAM (bank switched), an RS232 port, plus 5 interpreted languages and a 6809 assembler/editor system all written by the University of Waterloo. Aside from its obvious use as an educational tool, the rights were sold to Commodore for the purpose of marketing it as a highly powered business machine. By all indications it would have done well at the time, but Commodore, in their often typical brilliance, put it on hold in favor of pushing their now famous Protecto special, the B machine. They stopped a great computer from moving, to wait for a computer that they never moved. Reverse Commodore logic. And so, on with the story.

Chuck Peddle knew that the key to the future was in the design of a lower cost 8 bit chip that would appeal to a mass market. He felt that if the 6809 could be powered down, thus reducing the manufacturing cost, a winner would be born. Enter MOS Technology.

MOS was founded by a group of people who were far better at designing chips than they were at keeping the books. They quickly started in the design work of the 6500 series of chips, but just as quickly ran into financial problems. A great product without proper management to keep it afloat.

Enter stage left, Jack Tramiel. After the calculator wars in the mid 70's, Jack Tramiel was at a stage where Commodore was on some pretty shaky financial ground. In simple terms, the move Texas Instruments made to produce their own calculators and mass market them brought kaos to the calculator world as it was then known. When TI entered the calculator market, they brought with them a massive price reduction of their components. TI florished with high volume sales. Other manufacturers perished under the strain of competing against TI using older TI chips bought at much higher prices. The fatality rate was extremely high, with the majority of manufactures sinking due to inexperience and TI. At that time Commodore came pretty close to being one of the fatalities.

To Commodores rescue came Irving Gould, a very well to do financier. In exchange for bailing out Commodore, he received all of Jack Tramiels corporate stock, with the agreement that Jack Tramiel would get back a portion if he could get Commodore back on its feet. A sure bet for Irving Gould if he really knew Jack Tramiel.

Soon after the Commodore bail out, Jack Tramiel asked Irving Gould to back him in the purchase of MOS Technology, a good company in poor financial shape. The logic was that MOS had the capacity to do well, and could be bought for pennies on the dollar. With good management, Jack Tramiel was sure that MOS would make Commodore great. Never again was Jack Tramiel going to allow himself to be at the mercy of other manufacturers in the market place.

The balance is well known computer history. With incredible drive and determination, the team of Jack Tramiel and Chuck Peddle started Commodore on its path to glory. Beginning with the KIM microcomputer board, Commodore rapidly developed the home computer market as we know it today. And so, the majority of our history lesson has been completed.

If the past is any indication of future trends, Jack Tramiel is sure to bring Atari back into the world of the living. Something like the story of Frankenstein. Mad doctor Frankenstein worked like an animal salvaging people pieces here and there to create his monster. When the parts were assembled, and power was applied, presto, the creature was given life. The surprise is that it was more powerful than the sum of its parts, and just as unpredictable. Perhaps Atari, with the salvaged structure of Atari, and the brains of Commodore, will also produce a creature more powerful than the sum of its parts. Pure speculation.

To continue with the story, the 6500 series of chips have advanced very little in their true power. Although they now possess better memory management capabilities, it is still basically of the same eight bit design. Enter Motorola once again.

Unlike Chuck Peddles ideas regarding a power reduction of the Motorola chips, the people at Motorola could think of little else than increasing the chips capabilities. More power was the cry of the day, and so, a new chip was born. In a time when 8 bit was king, and 16 bits were a dream, the Motorola 68000 chip was considered revolutionary. Today, more than five years since its inception, the Motorola 68000 is one of the best. A totally new design without the limitations imposed by its 8 bit ancestor, the chip is incredible to say the least. A 16 bit data bus that can directly interface with existing 8 bit MC6800 peripherals, plus true 32 bit architecture that was designed to be a pleasure to program.

To avoid a long, drawn out rendition about how the 68000 will change your life, here is a quick synopsis of the 68000's special features:

1) Most instructions within its set apply to 8, 16, and 32 bit operations. All that is required is to specify the instruction with a suffix of .B for 8–Bit Byte, .W for 16–Bit Word, or .L for a 32–Bit Long Word.

2) There are eight 32–Bit data registers, and seven 32–Bit address registers at the programmers access.

3) Virtual memory access of 16 megabytes. (24 bits of 32)

4) Linear addressing in a standard 32 bit base.

5) It is a general–purpose register chip, therefore most instructions (eg. ADD) can be used for any combination of registers. The same instruction for all registers, just a change in the suffix of registers involved.

6) The MOVE instructions exist! In simple terms, a few incredible variations on the MOVE instruction allow data to be easily passed anywhere. Between registers, out ports, from ports, into memory, anywhere. To get you interested, there can be up to 34,888 combinations of MOVE made, for each of the 8, 16, and 32 Bit data types. Try that on a 6502!

To now remove the 68000 from the lime light, Motorola has announced the release of the 68010 chip, a totally compatible upgrade to the 68000. The sharp feature of the 68010 is that it has an upgraded access facility for up to 16 megabytes of virtual memory. Whatever is not RAM will be accessed from disk as virtual memory, with the processor going into a wait state until the contents from disk are brought into RAM. Once the virtual access is complete, processing continues. Along with the virtual memory access, a special bus access procedure has been further refined to allow faster bus access in a logical manner.

As a final salute to the progress of Motorola, another chip has been produced that most of us will never see. It's the 68020, a true 32 bit monster that operates with a clock speed of 12.5 MHz, soon to be 16.67 MHz. With a 32 bit bus and 32 bit architecture, it claims a speed increase over the 68000 of up to 400% in some instances. To further blow its horn, the maximum memory access capabilities have been increased from 16 megabytes to 4 gigabytes! Right now this would mean a mini or main frame, but give it a few years. The distinction between micro's, mini's, and main's is getting more difficult to determine every day. Another blatant speculation.

To once again return to the main subject matter, the MOS 6500 chips, and the Motorola 6800 and 68000 chips all share one thing: lineage. They were once related, therefore they share a similar instruction set. This is great news to the Commodore user. When, and if, Commodore releases the Amega Lorraine, it will be 68000 based. The Atari ST520 is also 68000 based. As a matter of fact, a quick look about the market will show that Intel and Motorola are basically the only ones involved in the business market. With the Atari 520 ST, it looks like the 68000 will make it into the home forum. Whatever the case, if you are at all interested in keeping up with todays trends, get to know the 68000. Future chips in the 68000 series will share the instruction set, so a bit of knowledge now will go a very long way.

Before advancing onto the hard core programming info, I would like to extend my sincere thanks to Robert Hamashuk, Field Applications Engineer with Motorola here in Toronto. Thanks to the research material he supplied, I have been able to go into much greater depth than ever anticipated regarding the Motorola chips. Thanks once again.

## MOS 6502 Registers:

| | | |
|---|---|---|
| A | Accumulator | : 8 Bit |
| X, Y | Index Registers | : 8 Bit |
| S | Stack Pointers | : 8 Bit Stack always held at $0100-$01FF |
| PC | Program Counter | : 16 Bit (Low/High) |
| P | Processor Status | : 8 Bits |

| | | | | | |
|---|---|---|---|---|---|
| Bit 0 | C | Carry Flag | Bit 4 | B | BRK Command |
| Bit 1 | Z | Result Zero | Bit 5 | x | Not In Use |
| Bit 2 | I | IRQ Disabled | Bit 6 | V | Overflow |
| Bit 3 | D | Decimal Mode | Bit 7 | N | Negative |

## Motorola 6809 Registers:

| | | |
|---|---|---|
| A, B, D | Accumulators | : D=16 Bits comprised of A+B (hi/lo) |
| X, Y | Index Registers | : 16 Bit |
| S, U | Stack Pointers | : 16 Bit : S=System Stack, U=User Stack |
| PC | Program Counter | : 16 Bit |
| DP | Direct Page | : 8 Bit |
| CC | Condition Code | : 8 Bits |

Bit 0 C Carry Flag
Bit 1 V Overflow Flag
Bit 2 Z Zero Flag
Bit 3 N Negative Flag
Bit 4 I Interrupt Request Flag
Bit 5 H Half Carry Flag (from bit 3)
Bit 6 F Fast Interrupt Flag
Bit 7 E Entire State Saved On Stack Flag

## Motorola 68000 Registers:

| | | |
|---|---|---|
| A0–A6 | Address Registers | : 32 Bit |
| D0–D7 | Data Registers | : 32 Bit |
| SSP | Stack Pointer | : 32 Bit Supervisor Stack A7 Addr Reg |
| USP | Stack Pointer | : 32 Bit User Stack A7 Addr Reg |
| PC | Program Counter | : 32 Bit Low Order 24 Bits In Use |
| SR | Status Register | : 16 Bits |
| CCR | Bits 0–7 of SR is the Condition Code Register | |

Bit 0 C Carry Flag
Bit 1 V Overflow Flag
Bit 2 Z Zero Flag
Bit 3 N Negative Flag
Bit 4 X Extend (similar to carry)
Bit 5 x Reserved Bit
Bit 6 x Reserved Bit
Bit 7 x Reserved Bit
Bits 8–15 of SR is the System Byte
Bit 8 I0 Interrupt Mask #1
Bit 9 I1 Interrupt Mask #2
Bit 10 I2 Interrupt Mask #3
Bit 11 x Reserved Bit
Bit 12 x Reserved Bit
Bit 13 S Supervisor State
Bit 14 x Reserved Bit
Bit 15 T Trace Mode

Note: SSP and USP are never active at the same time, thus they can 'share' register A7.

## 6502 Data Addressing Modes

01) Memory Immediate
02) Memory Absolute or Direct
03) Memory Zero Page (direct)
04) Implied or Inherent
05) Accumulator
06) Pre-Indexed Indirect
07) Post-Indexed Indirect
08) Zero Page Indexed
09) Absolute Indexed
10) Relative
11) Indirect

## 6809 Data Addressing Modes

01) Inherent
02) Accumulator
03) Immediate
04) Absolute a) ~~~~
05) Register
04) Absolute
 a) [direct]
 b) Extended
 c) Extended Indirect
05) Register
06) Indexed
 a) Constant-Offset Indexed
 b) Constant-Offset Indexed Indirect
 c) Accumulator Indexed
 d) Accumulator Indexed Indirect
 e) Auto-Increment
 f) Auto-Increment Indirect
 g) Auto-Decrement
 h) Auto-Decrement Indirect
07) Relative
08) Long Relative

## 68000 Data Addressing Modes

| Mode | Generation |
|---|---|
| Register Direct Addressing | |
| Data Register Direct | EA = Dn |
| Address Register Direct | EA = An |
| Absolute Data Addressing | |
| Absolute Short | EA = Next Word |
| Absolute Long | EA = Next Two Words |
| Program Counter Relative Addressing | |
| Relative With Offset | EA = (PC)+d16 |
| Relative With Index And Offset | EA = (PC)+(Xn)+d8 |
| Register Indirect Addressing | |
| Register Indirect | EA = (An) |
| Postincrement Register Indirect | EA = (An),An < An+N |
| Predecrement Register Indirect | An < An-N, EA = (An) |
| Register Indirect With Offset | EA = (An)+d16 |
| Indexed Register Indirect With Offset | EA = (An)+(Xn)+d8 |
| Immediate Data Addressing | |
| Immediate | DATA = Next Word(s) |
| Quick Immediate | Inherent Data |
| Implied Addressing | |
| Implied Register | EA = SR, USP, SSP, PC, VBR, SFC, DFC |

**Notes:**

| | | | |
|---|---|---|---|
| EA | = Effective Address | < | = Replaces |
| Dn | = Data Register | An | = Address Register |
| PC | = Program Counter | SR | = Status Register |
| | | () | = Contents Of |

Xn = Address Or Data Register Used As Index Register
d8 = 8-Bit Offset (Displacement)
d16 = 16-Bit Offset (Displacement)
N = 1 for byte, 2 for word, and 4 for long word. If An is the stack pointer and the operand size is byte, N=2 to keep the stack pointer on a word boundary.

# Instruction Set Comparison

## The MOS 6502, and Motorola 6809 and 68000 Chips

| Instr. | 6502 | 6809 | 68000 | Description |
|---|---|---|---|---|
| ABCD | | | ✓ | Add Decimal With Extend |
| ABX | | ✓ | | Add Accumulator B (unsigned) To Index Reg X |
| ADC | ✓ | | | Add Memory To Accumulator With Carry |
| ADCA | | ✓ | | Add Carry Bit And Memory Byte To Accum. A |
| ADCB | | ✓ | | Add Carry Bit And Memory Byte To Accum. B |
| ADD | | | ✓ | Add Binary |
| ADDA | | ✓ | | Add Memory Byte To Accumulator A |
| ADDA | | | ✓ | Add Address |
| ADDB | | ✓ | | Add Memory Byte To Accumulator B |
| ADDD | | ✓ | | Add 16 Bits Of Memory To Accumulator D |
| ADDI | | | ✓ | Add Immediate |
| ADDQ | | | ✓ | Add Quick |
| ADDX | | | ✓ | Add Extended |
| AND | ✓ | | ✓ | Logical AND |
| ANDA | | ✓ | | Logical AND Memory Byte To Accumulator A |
| ANDB | | ✓ | | Logical AND Memory Byte To Accumulator B |
| ANDCC | | ✓ | | Logical AND Memory Immediate Byte To CC Reg |
| ANDI | | | ✓ | Logical AND Immediate |
| ANDI to CCR | | | ✓ | Logical AND Immediate To Condition Codes |
| ANDI to SR | | | ✓ | Logical AND Immediate To Status Register |
| ASL | ✓ | ✓ | ✓ | Arithmetic Bit Shift Left |
| ASLA | | ✓ | | Arithmetic Bit Shift Left Accumulator A |
| ASLB | | ✓ | | Arithmetic Bit Shift Left Accumulator B |
| ASR | | ✓ | ✓ | Arithmetic Shift Right |
| ASRA | | ✓ | | Arithmetic Shift Right Accumulator A |
| ASRB | | ✓ | | Arithmetic Shift Right Accumulator B |
| BCC | ✓ | ✓ | ✓ | Branch On Carry Clear |
| BCHG | | | ✓ | Bit Test And Change |
| BCLR | | | ✓ | Bit Test And Clear |
| BCS | ✓ | ✓ | ✓ | Branch On Carry Set |
| BEQ | ✓ | ✓ | ✓ | Branch On Equal |
| BGE | | ✓ | ✓ | Branch On Greater Than or Equal |
| BGT | | ✓ | ✓ | Branch On Greater Than |
| BHI | | ✓ | ✓ | Branch On High |
| BHS | | ✓ | | Branch On Higher Or The Same |
| BIT | ✓ | | | Test Bits In Memory With Accumulator |
| BITA | | ✓ | | Bit Test - ANDing Memory Byte With Accum. A |
| BITB | | ✓ | | Bit Test - ANDing Memory Byte With Accum. B |
| BKPT | | | ✓ | Break Point |
| BLE | | ✓ | ✓ | Branch On Less Than Or Equal |
| BLO | | ✓ | | Branch On Lower |
| BLS | | ✓ | ✓ | Branch On Lower Or The Same |
| BLT | | ✓ | ✓ | Branch On Less Than |
| BMI | ✓ | ✓ | ✓ | Branch On Minus |
| BNE | ✓ | ✓ | ✓ | Branch On Not Equal |
| BPL | ✓ | ✓ | ✓ | Branch On Plus |
| BRA | | ✓ | ✓ | Branch Always |
| BRK | ✓ | | | Force Break |
| BRN | | ✓ | | Branch Never |
| BSET | | | ✓ | Test A Bit And Set |
| BSR | | ✓ | ✓ | Branch To Subroutine |
| BTST | | | ✓ | Test A Bit |
| BVC | ✓ | ✓ | ✓ | Branch On Overflow Clear |
| BVS | ✓ | ✓ | ✓ | Branch On Overflow Set |
| CHK | | | ✓ | Check Register Against Bounds |
| CLC | ✓ | | | Clear Carry Bit |
| CLD | ✓ | | | Clear Decimal Mode |
| CLI | ✓ | | | Clear Interrupt Disable |
| CLR | | ✓ | | Clear Memory Byte |
| CLR | | | ✓ | Clear An Operand |
| CLRA | | ✓ | | Clear Accumulator A |
| CLRB | | ✓ | | Clear Accumulator B |
| CLV | ✓ | | | Clear Overflow Bit |
| CMP | ✓ | | ✓ | Compare |
| CMPA | | ✓ | | Compare Memory Byte To Accumulator A |
| CMPA | | | ✓ | Compare Address |
| CMPB | | ✓ | | Compare Memory Byte To Accumulator B |
| CMPD | | ✓ | | Compare 16 Bits Of Memory To A 16 Bit Register |
| CMPI | | | ✓ | Compare Immediate |
| CMPM | | | ✓ | Compare Memory |
| CMPS | | ✓ | | Compare 16 Bits Of Memory To Stack Pointer |
| CMPU | | ✓ | | Compare 16 Bits Of Memory To User Stack Pointer |
| CMPX | | ✓ | | Compare 16 Bits Of Memory To X Register |
| CMPY | | ✓ | | Compare 16 Bits Of Memory To Y Register |
| COM | | ✓ | | Complement Accumulator Or Memory |
| COMA | | ✓ | | Complement Accumulator A Or Memory |
| COMB | | ✓ | | Complement Accumulator B Or Memory |
| CPX | ✓ | | | Compare Index Register X |
| CPY | ✓ | | | Compare Index Register Y |
| CWAI | | ✓ | | Clear And Wait For Interrupt |
| DAA | | ✓ | | Decimal Addition Adjust On Accumulator A |
| DBCC | | | ✓ | Decrement And Branch On Carry Clear |
| DBCS | | | ✓ | Decrement And Branch On Carry Set |
| DBEQ | | | ✓ | Decrement And Branch On Equal |
| DBF | | | ✓ | Decrement And Branch On Never True (False) |
| DBGE | | | ✓ | Decrement And Branch On Greater Than or Equal |
| DBGT | | | ✓ | Decrement And Branch On Greater Than |
| DBHI | | | ✓ | Decrement And Branch On High |
| DBLE | | | ✓ | Decrement And Branch On Less Than Or Equal |
| DBLS | | | ✓ | Decrement And Branch On Low Or The Same |
| DBLT | | | ✓ | Decrement And Branch On Less Than |
| DBMI | | | ✓ | Decrement And Branch On Minus |
| DBNE | | | ✓ | Decrement And Branch On Not Equal |
| DBPL | | | ✓ | Decrement And Branch On Plus |
| DBT | | | ✓ | Decrement And Branch On Always True |
| DBVC | | | ✓ | Decrement And Branch On Overflow Clear |
| DBVS | | | ✓ | Decrement And Branch On Overflow Set |
| DEC | ✓ | ✓ | | Decrement Memory By One |
| DECA | | ✓ | | Decrement Accumulator A By One |
| DECB | | ✓ | | Decrement Accumulator B By One |
| DEX | ✓ | | | Decrement The X Register |
| DEY | ✓ | | | Decrement The Y Register |
| DIVS | | | ✓ | Signed Divide |
| DIVU | | | ✓ | Unsigned Divide |
| EOR | ✓ | | ✓ | Exclusive OR Logical |
| EORA | | ✓ | | Exclusive OR Memory Byte To Accumulator A |
| EORB | | ✓ | | Exclusive OR Memory Byte To Accumulator B |
| EORI | | | ✓ | Exclusive OR Immediate |
| EORI to CCR | | | ✓ | Exclusive OR Immediate To Condition Codes |
| EORI to SR | | | ✓ | Exclusive OR Immediate To Status Register |
| EXG | | ✓ | ✓ | Exchange Registers |
| EXT | | | ✓ | Sign Extend |
| INC | ✓ | ✓ | | Increment Memory By One |
| INCA | | ✓ | | Increment Accumulator A By One |
| INCB | | ✓ | | Increment Accumulator B By One |
| INX | ✓ | | | Increment The X Register |
| INY | ✓ | | | Increment The Y Register |
| JMP | ✓ | ✓ | ✓ | Jump |
| JSR | ✓ | ✓ | ✓ | Jump To Subroutine |
| LBCC | | ✓ | | Long Branch On Carry Bit Clear |
| LBCS | | ✓ | | Long Branch On Carry Bit Set |
| LBEQ | | ✓ | | Long Branch On Equal |
| LBGE | | ✓ | | Long Branch On Greater Than Or Equal To Zero |
| LBGT | | ✓ | | Long Branch On Greater Than Zero |
| LBHI | | ✓ | | Long Branch On Higher |
| LBHS | | ✓ | | Long Branch On Higher Or The Same |
| LBLE | | ✓ | | Long Branch On Less Than Or Equal To Zero |
| LBLO | | ✓ | | Long Branch On Lower |
| LBLS | | ✓ | | Long Branch On Lower Or The Same |
| LBLT | | ✓ | | Long Branch On Less Than Zero |
| LBMI | | ✓ | | Long Branch On Minus |
| LBNE | | ✓ | | Long Branch On Not Equal |
| LBPL | | ✓ | | Long Branch On Plus |
| LBRA | | ✓ | | Long Branch Always |
| LBRN | | ✓ | | Long Branch Never |

| Instr. | 6502 | 6809 | 68000 | Description |
|---|---|---|---|---|
| LBSR | | ✓ | | Long Branch To Subroutine |
| LBVC | | ✓ | | Long Branch On Overflow Bit Clear |
| LBVS | | ✓ | | Long Branch On Overflow Bit Set |
| LDA | ✓ | | | Load Memory Byte Into Accumulator |
| LDA | | ✓ | | Load Memory Byte Into Accumulator A |
| LDB | | ✓ | | Load Memory Byte Into Accumulator B |
| LDD | | ✓ | | Load 16 Bits Of Memory In Accumulator D |
| LDS | | ✓ | | Load 16 Bits Of Memory In Stack Pointer |
| LDU | | ✓ | | Load 16 Bits Of Memory In User Stack Pointer |
| LDX | ✓ | | | Load 8 Bits Of Memory Into X Register |
| LDX | | ✓ | | Load 16 Bits Of Memory Into X Register |
| LDY | ✓ | | | Load 8 Bits Of Memory Into Y Register |
| LDY | | ✓ | | Load 16 Bits Of Memory Into Y Register |
| LEA | | | ✓ | Load Effective Address |
| LEAS | | ✓ | | Load Effective Address Into Stack Pointer |
| LEAU | | ✓ | | Load Effective Address Into User Stack Pointer |
| LEAX | | ✓ | | Load Effective Address Into X Register |
| LEAY | | ✓ | | Load Effective Address Into Y Register |
| LINK | | | ✓ | Link Stack And Allocate |
| LSL | | ✓ | ✓ | Logical Bit Shift Left Memory |
| LSLA | | ✓ | | Logical Bit Shift Left Accumulator A |
| LSLB | | ✓ | | Logical Bit Shift Left Accumulator B |
| LSR | ✓ | ✓ | ✓ | Logical Bit Shift Right Memory |
| LSRA | | ✓ | | Logical Bit Shift Right Accumulator A |
| LSRB | | ✓ | | Logical Bit Shift Right Accumulator B |
| MOVE | | | ✓ | Move Source To Destination |
| MOVEA | | | ✓ | Move Address |
| MOVEC | | | ✓ | Move To/From Control Register |
| MOVEM | | | ✓ | Move Multiple Registers |
| MOVEP | | | ✓ | Move Peripheral Data |
| MOVES | | | ✓ | Move To/From Address Space |
| MOVEQ | | | ✓ | Move Quick |
| MOVE from CCR | | | Yes | Move From Condition Codes |
| MOVE to CCR | | | ✓ | Move To Condition Codes |
| MOVE from SR | | | ✓ | Move From Status Register |
| MOVE to SR | | | ✓ | Move To Status Register |
| MOVE USP | | | ✓ | Move User Stack Pointer |
| MUL | | ✓ | | Multiply (unsigned) Accumulators A and B |
| MULS | | | ✓ | Signed Multiply |
| MULU | | | ✓ | Unsigned Multiply |
| NBCD | | | ✓ | Negate Decimal With Extend |
| NEG | | ✓ | ✓ | Negate Memory |
| NEGA | | ✓ | | Negate Accumulator A |
| NEGB | | ✓ | | Negate Accumulator B |
| NEGX | | | ✓ | Negate With Extend |
| NOP | ✓ | ✓ | ✓ | No Operation |
| NOT | | | ✓ | Logical Complement |
| OR | | | ✓ | Inclusive OR |
| ORA | ✓ | | | Logical OR Memory With Accumulator |
| ORA | | ✓ | | Inclusive OR Memory Immediate Byte To Accum A |
| ORB | | ✓ | | Inclusive OR Memory Immediate Byte To Accum B |
| ORCC | | ✓ | | Inclusive OR Memory Immediate Byte To CC Reg |
| ORI | | | ✓ | Inclusive OR Immediate |
| ORI to CCR | | | ✓ | Inclusive OR Immediate To Condition Codes |
| ORI to SR | | | ✓ | Inclusive OR Immediate To Status Register |
| PEA | | | ✓ | Push Effective Address |
| PHA | ✓ | | | Push The Accumulator Onto The Stack |
| PHP | ✓ | | | Push The Processor Status Onto The Stack |
| PLA | ✓ | | | Pull The Accumulator From The Stack |
| PLP | ✓ | | | Pull The Processor Status From The Stack |
| PSHS | | ✓ | | Push Specified Registers Onto System Stack |
| PSHU | | ✓ | | Push Specified Registers Onto User Stack |
| PULS | | ✓ | | Pull Specified Registers From System Stack |
| PULU | | ✓ | | Pull Specified Registers From User Stack |
| RESET | | | ✓ | Reset External Devices |
| ROL | ✓ | | | Rotate Bits Left Accumulator |
| ROL | | ✓ | | Rotate Bits Left Memory |
| ROL | | | ✓ | Rotate Bits Left Without Extend |
| ROLA | | ✓ | | Rotate Bits Left Accumulator A |
| ROLB | | ✓ | | Rotate Bits Left Accumulator B |
| ROR | ✓ | | | Rotate Bits Right Accumulator |
| ROR | | ✓ | | Rotate Bits Right Memory |
| ROR | | | ✓ | Rotate Bits Right Without Extend |
| RORA | | ✓ | | Rotate Bits Right Accumulator A |
| RORB | | ✓ | | Rotate Bits Right Accumulator B |
| ROXL | | | ✓ | Rotate Bits Left With Extend |
| ROXR | | | ✓ | Rotate Bits Right Without Extend |
| RTD | | | ✓ | Return And Deallocate Parameters |
| RTE | | | ✓ | Return From Exception |
| RTI | ✓ | ✓ | | Return From Interrupt |
| RTR | | | ✓ | Return And Restore Condition Codes |
| RTS | ✓ | ✓ | ✓ | Return From Subroutine |
| SBC | ✓ | | | Subtract Memory From Accumulator With Borrow |
| SBCA | | ✓ | | Subtract Carry Bit And Memory Byte From Accum A |
| SBCB | | ✓ | | Subtract Carry Bit And Memory Byte From Accum B |
| SBCD | | | ✓ | Subtract Decimal With Extend |
| SCC | | | ✓ | Set Conditional Byte Carry Clear |
| SCS | | | ✓ | Set Conditional Byte Carry Set |
| SEC | ✓ | | | Set Carry Bit |
| SED | ✓ | | | Set Decimal Mode |
| SEI | ✓ | | | Set Interrupt Disable |
| SEQ | | | ✓ | Set Conditional Byte Equal |
| SEX | | ✓ | | Sign Extended |
| SF | | | ✓ | Set Conditional Byte Never True (False) |
| SGE | | | ✓ | Set Conditional Byte Greater Than or Equal |
| SGT | | | ✓ | Set Conditional Byte Greater Than |
| SHI | | | ✓ | Set Conditional Byte High |
| SLE | | | ✓ | Set Conditional Byte Less Than Or Equal |
| SLS | | | ✓ | Set Conditional Byte Low Or The Same |
| SLT | | | ✓ | Set Conditional Byte Less Than |
| SMI | | | ✓ | Set Conditional Byte Minus |
| SNE | | | ✓ | Set Conditional Byte Not Equal |
| SPL | | | ✓ | Set Conditional Byte Plus |
| ST | | | ✓ | Set Conditional Byte Always True |
| STA | ✓ | | | Store Accumulator Into Memory Byte |
| STA | | ✓ | | Store Accumulator A Into Memory Byte |
| STB | | ✓ | | Store Accumulator B Into Memory Byte |
| STD | | ✓ | | Store Accumulator D Into 16 Bit Memory Location |
| STOP | | | ✓ | Load Status Register And Stop |
| STS | | ✓ | | Store Stack Pointer Into 16 Bit Memory Location |
| STU | | ✓ | | Store User Stack Ptr. Into 16 Bit Memory Location |
| STX | ✓ | | | Store X Register Into 8 Bit Memory Location |
| STX | | ✓ | | Store X Register Into 16 Bit Memory Location |
| STY | ✓ | | | Store Y Register Into 8 Bit Memory Location |
| STY | | ✓ | | Store Y Register Into 16 Bit Memory Location |
| SUB | | | ✓ | Subtract Binary |
| SUBA | | ✓ | | Subtract Memory Byte From Accumulator A |
| SUBA | | | ✓ | Subtract Address |
| SUBB | | ✓ | | Subtract Memory Byte From Accumulator B |
| SUBD | | ✓ | | Subtract 16 Bits Of Memory From Accumulator D |
| SUBI | | | ✓ | Subtract Immediate |
| SUBQ | | | ✓ | Subtract Quick |
| SUBX | | | ✓ | Subtract With Extend |
| SVC | | | ✓ | Set Conditional Byte Overflow Clear |
| SVS | | | ✓ | Set Conditional Byte Overflow Set |
| SWAP | | | ✓ | Swap Data Register Halves |
| SWI | | ✓ | | Software Interrupt #1 |
| SWI2 | | ✓ | | Software Interrupt #2 |
| SWI3 | | ✓ | | Software Interrupt #3 |
| SYNC | | ✓ | | Syncronize To External Event |
| TAS | | | ✓ | Test And Set Operand |
| TAX | ✓ | | | Transfer The Accumulator Into The X Register |
| TAY | ✓ | | | Transfer The Accumulator Into The Y Register |
| TFR | | ✓ | | Transfer Register To Register |
| TRAP | | | ✓ | Trap |
| TRAPV | | | ✓ | Trap On Overflow |
| TST | | ✓ | | Test Memory |
| TST | | | ✓ | Test An Operand |
| TSTA | | ✓ | | Test Accumulator A |
| TSTB | | ✓ | | Test Accumulator B |
| TSX | ✓ | | | Transfer The Stack Pointer Into The X Register |
| TXA | ✓ | | | Transfer The X Register Into The Accumulator |
| TXS | ✓ | | | Transfer The X Register Into The Stack Pointer |
| TYA | ✓ | | | Transfer The Y Register Into The Accumulator |
| UNLK | | | ✓ | Unlink |

# The Intel 8088 Microprocessor

Richard Evers, Editor

Back in the days of olde, circa 1981, IBM released their IBM PC complete with an Intel 8088 microprocessor. The 8088 is unique in that it has 16 bit architecture with an 8 bit data bus. The 8 bit bus was incorporated to allow a fast acceptance into the market due to the high proliferation of 8 bit support chips available at the time. Although a truly fast brother to the 8088 was available, the Intel 8086, the 8088 was chosen. IBM traded off speed for quick market entry.

Star Date 1985: Commodore announces the Commodore PC–10 and PC–20, IBM clones with a difference. Better pricing, complete compatibility, and a few nice hardware features standard. The trick is that they tried too hard to be compatible. The Intel 8088 is still there! Intel now has the 80186 and 80286, which are 8086's with power to spare. They share the same instruction set as the 8086/8088 chips, but have all sorts of extras on board which make the 8086 look archaic. IBM has released the IBM AT, which comes with an 80286 microprocessor on board. The neat trick with this one is that software written for the normal PC will execute just fine, but with an incredible increase in speed.

To follow up on this trend, clones such as the Compaq have followed suit, using an 80286 monster that runs with an 8 megahertz clock. This one's so incredible that you can get into multi–tasking at two different clock speeds! One use for two separate clock speeds is in using the Intel 8087 Numeric Data Processer. This chip has the capacity to perform all math functions (80 bit) via hardware, at a speed unsurpassed by anything but a mainframe. The trick is, this chip runs at 5 megahertz. It also has to run concurrent with the computer's main processor, ie. also at 5 megahertz. Normally, if you had a fast chip and you wanted to use the 8087 for number crunching, you would have to slow down the main processor to match. With the 80286 this is not so. One half of the multi–tasking environment can work at the same speed of the 8087, the other half can operate at top speed. No trade–off of speed for special functions.

This little bit of IBM hype is intended to demonstrate the typical 'too little, too late' philosophy common to most of the cloners as well as Commodore. The PC–10 and PC–20 are great machines, just as most clones are. They are truly compatible, and do have some terrific features that most don't come with. And the pricing is fine. But it's a shame to suffer in the capacity of the machine just to clone a standard IBM PC. A trade–off in speed and capability to be able to state that it is truly compatible is an odd way to enter into the market four years too late. Just think how nice it would have been to read some Commodore propaganda stating that their clone was software compatible, but able to outperform the standard IBM PC, 10 to 1. The Compaq can state this without fear of retraction.

To finally get on track for the balance of this article, I would like to introduce you to the Intel 8086/8088 instruction set, a nice treat in the programming department. As stated earlier, the 8086 and 8088 share an identical instruction set, but the 8086, with its 16 bit data bus, can move data at a much faster rate. No matter, they are nice processors to work with.

If you are at all familiar with the MOS 6502 series of chips, you know their philosophy of storing all data in a low/high fashion. Not all chip manufacturers do this. For example, Motorola's 6809 and MC68000 chips arrange their 16 bit words in high/low fashion, as does the Zilog Z80 and Z8000 chips. With the Intel chips, luck has it that they store their 16 bit words in low/high order. A point of trivia that the industry might try sorting out.

A feature that you will soon grow to appreciate with the Intel chips is their capacity to access 1 megabyte of RAM/ROM. The trick is called segmentation. Segmentation works through the use of two 16 bit registers. These registers are called the Segment Paragraph Address and the Offset. The real memory address is computed as such:

Real Address = 16 x Segment Paragraph Address + Offset

This is equivalent to a Shift Left on the Segment Paragraph Address, then adding in the Offset. Therefore, if the Segment Paragraph Address is set to $0500, and the Offset is $0200, then the address = $5000 + $0200 = $5200.

Depending on the operation performed, the Segment Paragraph Address is held in one of the following 16 bit segment registers:

| Word | Function |
|---|---|
| .CS | Code Segment |
| .DS | Data Segment |
| .SS | Stack Segment |
| .ES | Extra Segment |

The processor also has a number of interesting registers. The accumulator is really three registers. AX is the 16 bit accumulator, but AL and AH are the low/high bytes of the accumulator. In this way 16 bit or 8 bit operations can be easily performed.

Without taking up the entire issue to learn how to program the 8088, I am going to barrage you with 8088 info. To best understand the 8088's registers and operations, a good book may be a good investment. The '8086/8088 16–Bit Microprocessor Primer' by Christopher L. Morgan and Mitchell Waite was extra helpful for me. The book is extremely well written, and they go into good depth on all aspects of the Intel chips.

And so, on to a barrage of Intel info. Have a fine time.

Note: The term 'word' refers to 16 bit data in the following text.
The term 'byte' refers to the standard 8 bit byte.

### Intel 8086/8088 Registers:

| Word | Byte | Byte | |
|---|---|---|---|
| .AX | AH | AL | Accumulator |
| .BX | BH | BL | Base |
| .CX | CH | CL | Count |
| .DX | DH | DL | Data |
| .SP | | | Stack Pointer |
| .BP | | | Base Pointer |
| .SI | | | Source Index |
| .DI | | | Destination Index |
| .IP | | | Instruction Pointer |
| .CS | | | Code Segment |
| .DS | | | Data Segment |
| .SS | | | Stack Segment |
| .ES | | | Extra Segment |
| | SL | SH | Status Flags (see below) |

. Status Flags (bits)

| 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| xx | xx | xx | xx | OF | DF | IF | TF | SF | ZF | xx | AF | xx | PF | xx | CF |

| | |
|---|---|
| xx = not used | SF = Sign Flag |
| OF = Overflow Flag | ZF = Zero Flag |
| DF = Direction Flag (strings) | AF = Auxillary Carry – BCD |
| IF = Interrupt Enable Flag | PF = Parity Flag |
| TF = Trap – Single Step Flag | CF = Carry Flag |

### Intel 8088/8086 Instruction Set

## Data Transfer Instructions

| | | |
|---|---|---|
| MOV | dest, source | MOVe word from source to destination |
| MOVB | dest, source | MOVe Byte from source to destination |
| MOVI | dest, data | MOVe Immediate data to destination |
| MOVBI | dest, data | MOVe Immediate data to Byte destination |
| XCHG | dest, source | eXCHanGe contents of word locations |
| XCHGB | dest, source | eXCHanGe contents of Byte locations |
| PUSH | source | PUSH source onto stack |
| POP | dest | POP stack into destination |
| IN | source | INput from source to AX (word) |
| INB | source | INput from source to AL (Byte) |
| IN | | INput from location (DX) to AX (word) |
| INB | | INput from location (DX) to AL (Byte) |
| OUT | dest | OUTput from AX (word) to destination |
| OUTB | dest | OUTput from AL (Byte) to destination |
| OUT | | OUTput from AX (word) to location (DX) |
| OUTB | | OUTput from AL (Byte) to location (DX) |
| XLAT | | transLATe (using a tables) |
| LEA | register, source | Load Effective Address |
| LDS | register, source | Load DS and register |
| LES | register, source | Load ES and register |

## Binary Integer Arithmetic

| | | |
|---|---|---|
| NEG | dest | Take NEGative of destination |
| NEGB | dest | Take NEGative of Byte destination |
| ADD | dest, source | ADD source to destination (word) |
| ADDB | dest, source | ADD source to destination (Byte) |
| ADDI | dest, data | ADD Immediate data to destination (word) |
| ADDBI | dest, data | ADD Immediate data to destination (Byte) |
| ADC | dest, source | ADd source + Carry to destination (word) |
| ADCB | dest, source | ADd source + Carry to destination (Byte) |
| ADCI | dest, data | ADd data + Carry to destination |
| ADCBI | dest, data | ADd data + Carry to Byte destination |
| SUB | dest, source | SUBtract source from destination (word) |
| SUBB | dest, source | SUBtract source from destination (Byte) |
| SUBI | dest, data | SUBtract data from destination (word) |
| SUBBI | dest, data | SUBtract data from destination (Byte) |
| SUBB | dest, source | SUBtract source + Borrow from destination |
| SUBBB | dest, source | SUBtract source + Borrow from Byte dest. |
| SUBBI | dest, data | SUBtract data + Borrow from destination |
| SUBBBI | dest, data | SUBtract data + Borrow from Byte dest. |
| MUL | source | unsigned 16-bit MULtiply |
| MULB | source | unsigned 8-bit MULtiply |
| IMUL | source | signed 16-bit MULtiply |
| IMULB | source | signed 8-bit MULtiply |
| DIV | source | unsigned 16-bit DIVide |
| DIVB | source | unsigned 8-bit DIVide |
| IDIV | source | signed 16-bit DIVide |
| IDIVB | source | signed 8-bit DIVide |
| CBW | | Convert from Byte to Word |
| CWD | | Convert from Word to Byte |
| INC | dest | INCrement destination (word) |
| INCB | dest | INCrement destination (Byte) |
| DEC | dest | DECrement destination (word) |
| DECB | dest | DECrement destination (Byte) |

## Logical Operations

| | | |
|---|---|---|
| NOT | dest | take logical NOT of the destination (word) |
| NOTB | dest | take logical NOT of the destination (Byte) |
| AND | dest | logical AND of source and destination (word) |
| ANDB | dest | logical AND of source and destination (Byte) |
| ANDI | dest, data | logical AND of data and destination (word) |
| ANDBI | dest, data | logical AND of data and destination (byte) |
| OR | dest, source | logical OR of source and destination (word) |
| ORB | dest, source | logical OR of source and destination (Byte) |
| ORI | dest, data | logical OR of data and dest (word) |
| ORBI | dest, data | logical OR of data and destination (Byte) |
| XOR | dest, source | logical XOR of source and destination (word) |
| XORB | dest, source | logical XOR of source and destination (Byte) |
| XORI | dest, data | logical XOR of data and destination (word) |
| XORBI | dest, data | logical XOR of data and destination (Byte) |

## Shifts And Rotates

| | | |
|---|---|---|
| SHL | dest | logical SHift Left one bit (word) |
| SHL | dest, CL | logical SHift Left CL bits (word) |
| SHLB | dest | logical SHift Left one bit (Byte) |
| SHLB | dest, CL | logical SHift Left CL bits (Byte) |
| SHR | dest | logical SHift Right one bit (word) |
| SHR | dest, CL | logical SHift Right CL bits (word) |
| SHRB | dest | logical SHift Right one bit (Byte) |
| SHRB | dest, CL | logical SHift Right CL bits (Byte) |
| SAL | dest | Arithmetic Shift Left one bit (word) |
| SAL | dest, CL | Arithmetic Shift Left CL bits (word) |

| | | |
|---|---|---|
| SALB | dest | Arithmetic Shift Left one bit (Byte) |
| SALB | dest, CL | Arithmetic Shift Left CL bits (Byte) |
| SAR | dest | Arithmetic Shift Right one bit (word) |
| SAR | dest, CL | Arithmetic Shift Right CL bits (word) |
| SARB | dest | Arithmetic Shift Right one bit (Byte) |
| SARB | dest, CL | Arithmetic Shift Right CL bits (Byte) |
| ROL | dest | ROtate Left one bit (word) |
| ROL | dest, CL | ROtate Left CL bits (word) |
| ROLB | dest | ROtate Left one bit (Byte) |
| ROLB | dest, CL | ROtate Left CL bits (Byte) |
| ROR | dest | ROtate Right one bit (word) |
| ROR | dest, CL | ROtate Right CL bits (word) |
| RORB | dest | ROtate Right one bit (Byte) |
| RORB | dest, CL | ROtate Right CL bits (Byte) |
| RCL | dest | Rotate Left through Carry one bit (word) |
| RCL | dest, CL | Rotate Left through Carry CL bits (word) |
| RCLB | dest | Rotate Left through Carry one bit (Byte) |
| RCLB | dest, CL | Rotate Left through Carry CL bits (Byte) |
| RCR | dest | Rotate Right through Carry one bit (word) |
| RCR | dest, CL | Rotate Right through Carry CL bits (word) |
| RCRB | dest | Rotate Right through Carry one bit (Byte) |
| RCRB | dest, CL | Rotate Right through Carry CL bits (Byte) |

### String Manipulation

| | |
|---|---|
| REP | REPeat (used to modify next string instr.) |
| REPZ | REPeat while Zero |
| REPNZ | REPeat while Not Zero |
| REPE | REPeat while Equal |
| REPNE | REPeat while Not Equal |
| MOVC | MOVe string Characters (byte) |
| MOVW | MOVe string Words |
| CMPC | CoMPare string Characters (byte) |
| CMPW | CoMPare string Words |
| SCAC | SCan string Characters (byte) |
| SCAW | SCan string Words |
| LODC | LOaD string Characters (byte) |
| LODW | LOaD string Words |
| STOC | STOre string Characters (byte) |
| STOW | STOre string Words |
| CLD | Clear Direction flag |
| STD | SeT Direction flag |

### Program Control Operators

| | | |
|---|---|---|
| JMP | target | JuMP direct within segment |
| JMP | target, segment | JuMP direct to new segment |
| JMPS | dest | JuMp Short |
| JMPI | dest | JuMp Indirect within segment |
| JMPL | dest | JuMp Indirect Long (new segment) |
| JE | target | Jump if Equal |
| JZ | target | Jump if Zero |
| JNE | target | Jump if Not Equal |
| JNZ | target | Jump if Not Zero |
| JS | target | Jump if Sign (negative) |
| JNS | target | Jump if Not Sign (non–negative) |
| JP | target | Jump if Parity (parity even) |
| JNP | target | Jump if Not Parity (parity odd) |
| JPE | target | Jump if Parity Even |
| JPO | target | Jump if Parity Odd |
| JL | target | Jump if Less than |
| JNGE | target | Jump if Not Greater than or Equal to |
| JNL | target | Jump if Not Less than |
| JGE | target | Jump if Greater than or Equal to |
| JLE | target | Jump if Less than or Equal to |
| JNG | target | Jump if Not Greater than |
| JNLE | target | Jump if Not Less than or Equal to |
| JG | target | Jump if Greater than |
| JB | target | Jump if Below |
| JNAE | target | Jump if Not Above or Equal to |
| JNB | target | Jump if Not Below |
| JAE | target | Jump if Above or Equal to |
| JBE | target | Jump if Below or Equal to |
| JNA | target | Jump if Not Above |
| JNBE | target | Jump if Not Below or Equal to |
| | | |
| TEST | dest, source | TEST (word) |
| TESTB | dest, source | TEST (Byte) |
| TESTI | dest, data | TEST word against Immediate data |
| TESTBI | dest, data | TEST Byte against Immediate data |
| | | |
| CMP | dest, source | CoMPare word |
| CMPB | dest, source | CoMPare Byte |
| CMPI | dest, data | CoMPare word against Immediate data |
| CMPBI | dest, data | CoMPare Byte against Immediate data |
| | | |
| LOOP | target | LOOP |
| LOOPZ | target | LOOP if Zero |
| LOOPNZ | target | LOOP if Not Zero |
| LOOPE | target | LOOP if Equal |
| LOOPNE | target | LOOP if Not Equal |
| | | |
| JCXZ | target | Jump if CX is Zero |
| | | |
| CALL | target | CALL direct within segment |
| CALL | target, segment | CALL direct to new segment |
| CALLI | dest | CALL indirect within segment |
| CALLL | dest | CALL indirect Long (new segment) |
| | | |
| RET | | RETurn within segment |
| RET | number | RETurn within segment and adjust stack |
| RETS | | RETurn from segment |
| RETS | number | RETurn from segment and adjust stack |

### System Control

| | |
|---|---|
| INT | INTerrupt |
| INTO | INTerrupt if Overflow |
| IRET | Interrupt RETurn |
| CLI | CLear Interrupt flag |
| STI | SeT Interrupt Flag |
| | |
| HLT | HaLT the cpu |
| WAIT | WAIT (used to synco links of cpu with co–cpu) |
| | |
| LOCK | LOCKs bus on next instr. from access by other cpu |
| | |
| ESC | ESCape (calls a co–processor into action) |
| | |
| NOP | NO Operation |
| | |
| CLC | CLear Carry |
| STC | SeT Carry |
| CMC | CoMplement Carry |
| | |
| SAHF | Store AH into Flags |
| LAHF | Load AH from Flags |
| PUSHF | PUSH Flags |
| POPF | POP Flags |

# A Quick PC Primer

Richard Evers, Editor

### Commodore PC-10 File Formats

As many of you already know, there are four different disk file formats available for use with your standard Commodore drive. There is Sequential, Relative, Program, and User type files. Each have their own special merit in use, and each have been discussed at length in preceding issues. The purpose of todays article is bring about a bit of knowledge on the PC-10, the IBM PC clone from Commodore, and how it compares with currently available file formats.

To understand file formats a little more, you have to remember that all data stored on diskette is really sequential data, accessed a little differently by the ROM routines responsible. Sequential and User files are identical, with data written to and read back in the same manner, sequentially. Program files are also read through sequentially, but the first two bytes are special for the Loading procedure. Program files in the land of Commodore are handled specially due to the PRG extension in the Load department. Relative files are actually sequential data files that can be accessed by specific records at will. The data within the records can be read sequentially, but greater freedom is allowed by the use of side sectors for keeping track of the track and sectors involved and the ROM routines for calculating the indexing required. So much for normal file formats with normal Commodore machines.

The PC-10 does share all Commodore drive file formats of past. Sequential, User, Relative, and Program all exist. But the DOS does not put a special marking on the files to inform you of the data type within. This is up to the user.

Filenames in MS-DOS have a maximum length of 8 characters, and a maximum extension after the filename of 3 characters. The delimiter between filename and extension is a period. Any filename you can type in, with the exclusion of a few special characters or reserved extensions, are at your disposal. Without DOS automatically assigning all extensions, this leaves room for some pretty obtuse extensions if used without thought.

Program files, as created through the SAVE process in the MicroSoft BASIC supplied with the PC-10, are pretty interesting. You can SAVE a file as in normal program format, with a default extension by the system of .BAS, or you can SAVE the program in ASCII format, or you can SAVE it in a protected form. The ASCII format is used if you want to MERGE the program over top of another program you are working on. ASCII program files can also be LOADed and RUN as normal ones. Protected files are just program files that cannot be listed, at least without digging into RAM a bit to flip a few bits.

Relative files, called Random files in MS DOS speak, are identical in concept to Commodore Relative files. The big plus is that Random Files don't have the tiny cap on record size as normal Commodore Relative files do. With MS DOS, you can have a maximum record size of 32768 bytes. Commodore Relative records are maximum 254 bytes. With both types, aside for the Commodore 8050 drive, the maximum file size is restricted only by the room available on diskette.

There is an odd note to mention here about MS DOS file work. If you will be working with Random records in excess of 128 bytes, you have to set up the buffer size from DOS before booting up BASIC. Due to the fact that DOS is resident in computer RAM, all the file buffers are also. From within DOS, special things such as the maximum number of files Open at any time, the maximum size of each buffer, the size of the serial buffer, and a host of other equally thrilling parameters, should be thought of before booting up BASIC. Although the defaults of each are pretty logical, sometimes they just don't fit. Another point to remember when setting the parameters. The larger you go, and the more files you leave room to Open, the greater detraction from the 60k plus BASIC work space available. It won't affect many people, but it's a point to ponder.

To create a Random file is not a very difficult task. The following program will create a Random file, write 10 records of data, Close up the file, then re-Open and read through each record sequentially. Not a terribly exciting example, but it does show how Random files can be easily attained by the novice.

```
100 ' Random File Demo Program
105 OPEN "R",#1,"RANDOM.RND",100 ' Record Size Of 100 Bytes
110 FIELD#1,25 AS FIRST$, 25 AS SECOND$, 25 AS THIRD$,
    25 AS FOURTH$
115 FOR LOOP = 1 TO 10
120  LSET FIRST$ = STR$(LOOP)      ' Left Justify All Strings
125  LSET SECOND$ = STR$(LOOP*10)  ' Into Buffer For Write
130  LSET THIRD$ = STR$(LOOP*100)
135  LSET FOURTH$ = STR$(LOOP*1000)
140  PUT#1,LOOP ' Write Record In
145 NEXT LOOP
150 CLOSE#1
155 '
160 OPEN "R",#1,"RANDOM.RND"
165 FIELD#1,25 AS FIRST$, 25 AS SECOND$, 25 AS THIRD$,
    25 AS FOURTH$
170 FOR LOOP = 1 TO 10
175  GET#1,LOOP ' Get The Appropriate Record
180  PRINT FIRST$, SECOND$, THIRD$, FOURTH$
185 NEXT LOOP
190 CLOSE#1
```

For your own edification, the ' is another form of the REM statement to flag comments. REM does exist in MS BASIC, but the apostrophe is much tidier, in my opinion.

The example above is both in Upper and Lower case, with Lower case only appearing in comment lines. The reason is because the interpreter allows you to type everything in either case, but automatically converts all executable code into Upper case.

To start, line 105 Opens 'RANDOM.RND' for Random access, with a record length of 100 bytes. The "R" following the OPEN keyword signifies Random Access. For all other file formats, the "R" cannot be used. They have their own special indications for whatever file work is required.

Line #110 sets up the file buffer to accept the data for the write. In this example, the first 25 bytes in the file buffer will come from string variable FIRST$, the second 25 bytes from SECOND$, etc. Once a FIELD statement has been executed for a Read or Write, it remains the same for that particular logical file number. For this example, the logical file number assigned is #1.

The lines 105–145 loop through a procedure of assigning the correct string with test data, and moving the string data in the correct position within the buffer. LSET is a command to Left Justify the data into the buffer, padding with spaces as required. This command has a second cousin by the name of RSET. Predictably, it Right Justifies the data in the buffer.

When all data has been transferred into the buffer, a single PUT# statement is used to PUT the record #LOOP to disk. A fairly simple concept to grasp.

Line #150 Closes logical file #1 to end our write demo. The CLOSE statement can be used in a variety of ways. You can CLOSE one specific logical file, or a number of logical files via CLOSE #1,#2,#3, etc., subbing in the logical file #'s affected. If a single CLOSE was used, all currently OPEN files would be Closed up immediately.

Line #'s 160–190 perform the read the data in routine. The file is Opened once again for Random access, with the record length not being specified at the programmers discretion. The file buffer is set up accordingly through the FIELD statement, then the fun begins. Each record is read sequentially through the use of the GET# statement, with the strings thereafter being printed. Not a very difficult procedure, as most can see.

With Random access described in whole, Sequential access techniques begin. To create, write to, and read from sequential files is no major trick. Look below for a program that suits the occasion.

```
200 ' Sequential File Create/Read Routine
205 OPEN "O",#1,"SEQFILE.SEQ" ' Open File For Output (write)
210 FOR LOOP = 10 TO 20
215  PRINT#1,STR$(LOOP);CHR$(13);
220 NEXT LOOP
225 CLOSE#1
230 '
235 OPEN "I",#1,"SEQFILE.SEQ" ' Open File For Input (read)
240 INPUT#1,A$: PRINT A$: IF EOF(1) = 0 THEN 240 ELSE CLOSE#1
```

Line #205 shows a standard OPEN statement, this time using a "O" to indicate an Output (Write) procedure. Lines 210–225 write 10 sets of test data to the file, then Closes it up. The PRINT# statement can be replaced by a whole slew of commands to suit your needs. PRINT# USING exists as does WRITE#, for the purpose of formatting the output generated. No more special string work required for all who like nice looking, formatted files. Microsoft to the rescue.

Line 235 Opens the file once again, this time for an Input (Read) Operation. The "I" is the flag for this procedure. Line 240 Inputs and Prints all the data held in the file. The function EOF(1) flags the user when the end of file has been reached by returning a value of –1. When this happens, the ELSE statement comes into play thus Closing up the file.

To further entice you, another replacement has been invented for the ever bugged up INPUT# statement. The INPUT$ statement. INPUT# is still stopped by delimiters such as the comma, carriage return, and colon. INPUT$ is not. The format of INPUT$ is as follows:

```
A$ = INPUT$(numchar,logadd)
```

. . .where numchar is the number of characters to read each time, and logadd is the logical file address to read from. If the logadd is left off, the default will be from the keyboard.

Another feature exists with the PC–10 that has always been a favourite with Commodore DOS users. The Append feature. By Opening a sequential file with "A", you can write directly to the end of the file. In reality, Commodore DOS and MS DOS are not that far apart in concept. Commodore DOS is more automatic and user friendly, but MS DOS has extra advantages such as greater speed and versatility due to DOS upgrades without surgery to the drive.

There are a number of different extensions that the system will automatically assign to filenames of various orgins. They are the system files of DOS, batch processing files, and a host of other file types. The .BAT or batch file will be discussed next, but if the MS bug has really hit you, your best option would be to invest in a few of the PC magazines available, and hunt around for a book or two on the subject.

### Batch Processing With Your PC–10

The Commodore PC–10, the IBM PC compatible machine, is vastly different from any machine Commodore has released before. The Commodore of past has always prided itself in marketing their own designs. The microprocessors were always of MOS design, the architecture always typical Commodore, absolutely everything had a typical Commodore feel. Well, with the PC–10, Commodore has finally accepted that *same* is easier than *different*. The PC–10 is an IBM PC clone, with a few improvements. The keyboard is nicer to use, the standard options have been enlarged, and the price is also significantly lower. A clone to be proud of.

With the new Commodore machine on the scene, a whole new mind set will be required for those uninitiated with the IBM PC. The drives are no longer intelligent, therefore the DOS has to be loaded into computer RAM before access to the drive can really begin. The BASIC language is no longer resident in ROM, therefore BASIC, or some other language, will also have to be brought in from disk after DOS. But, even with these tedious shortcomings, a breath of fresh air appears. The entire booting up process on system initialization can be automatically performed with little effort, allowing the DOS, system parameters, language, and first program to all be brought in or set up as required. Welcome to Batch Processing, a welcome friend in a strange new land.

For anyone familiar with the Power command EXEC, or Chris Zamara's STP from a few issues back, the concept of operation is similar. They all allow you to create a sequential file on diskette that can be read from and executed by the computer as if entered directly from the keyboard. This allows you to perform some pretty terrific procedures on a repetitive basis without the major keyboard hassles.

When the PC-10 is first powered up, or re-booted via (Control) (Alt) (Del), the DOS is automatically brought in from the default drive, normally drive A, then a file by the name of 'AUTOEXEC-.BAT' is checked for. If it exists then the file is read through and executed sequentially. If the file is not found, the system drops into normal DOS mode.

The 'AUTOEXEC.BAT' file is a batch file with a special name. Batch files can be easily created that will batch process your needs, but 'AUTOEXEC.BAT' is the only one capable of executing from system start.

To create the Autoexec file from DOS, little work is required. From within DOS, type in the following:

```
COPY CON A:AUTOEXEC.BAT (Carriage Return)
```

The Drive A has been specified in this example. Drive A is the upper drive on the unit, drive B the lower.

What this command does is tell DOS to copy the following information from the keyboard into the Autoexec file, until a (Control z) is encountered. In this manner, any sequential file desired can be easily created.

Try typing in the following sequence of commands as described below

```
DATE         (Carriage Return)
TIME         (Carriage Return)
BASICA       (Carriage Return)
(Control z)  (Carriage Return)
```

The (Control z) followed by a carriage return will terminate the session, and tell DOS to write the file to diskette. Once this file has been executed by the system upon initialization, the system will prompt the user for the date, defaulting to January 1st 1980, as per IBM format, then the time. Following the correct replies from the user, carriage returns or the correct date and time, the language BASICA will be loaded into memory and executed.

If you wanted to load and run a specific program after BASICA, then modify the BASICA line as follows:

```
BASICA FILENAME.BAS (Carriage Return)
```

Filenames in IBM land have a maximum length of 8 characters, with an extension after a period of 3 characters maximum. If the program to be loaded has an extension of .BAS, indicating a BASIC program file, then it does not have to be specified in the Autoexec file. BASICA will automatically default to an extension of .BAS when Loading and Saving to disk.

Often, special tricks have to be performed via the Autoexec file to set up the computer as your program requires. The maximum number of files allowed open at any time, the size of the file buffer, maximum 32768 bytes, the size of the serial buffer, and the mode of display are just a few of the parameters to be chosen. The system defaults to logical choices, but often when writing business software special parameters will be required.

Although special emphasis has been placed on the Autoexec batch file, normal batch files can be pretty important too. Batch files can be created to execute special functions such as LOADing and executing programs of special importance simply by keying in a simple filename. Take for example the program Lotus 1-2-3. In DOS mode, execution of Lotus is done by keying in the name 'lotus', followed by a carriage return. There is a batch file on diskette by the name 'lotus' that fires up the program automatically for you. The same applies for most commercial software packages available for the IBM PC. They have Autoexec batch files used for system start up, and they also have an easily remembered file name for start up from DOS without (Control) (Alt) (Del). Made simple for the business market.

This article has been written as a very simple batch processing tutorial for those just getting into MS DOS, and does not make the disclaimer of trying to inform you of all the special tricks batch files can perform. It is just a method to get the ball rolling for IBM PC mindset to set in. To really get to know your DOS, read through a few of the many MS or PC DOS books on the market. Some are pretty poor, but a few will shine through. If you actually have the PC-10, or some other MS DOS machine, then read through the manuals supplied. Though the manuals tend to be brief, knowledge can be attained for the price of a little time.

So much for force feeding you DOS. Following this is a summary of DOS and BASIC commands that I hope may one day come in handy for you.

And lastly, although the PC-10 is a powerful machine, it is an IBM PC clone that will not be making a regular appearance in the pages of The Transactor. Placing the Commodore label on the machine does not justify using precious magazine space, especially considering the other publications dedicated solely to this system. Life was so much simpler when Commodore was Commodore.

# Commodore PC-10 Microsoft BASIC Command Summary

You will find that most of the keywords are identical to Commodore BASIC, plus many more just to keep your programming hours productive. With an equivalent of 175 commands at your access, sleepless nights will soon become a reality. Without further delay, welcome to your nightmare !!

| Command | Type | Description |
|---|---|---|
| ABS | Func | Returns absolute value |
| AND | | Boolean: x AND y = 1 if x,y=1, otherwise=0 |
| ASC | Func | Returns the ASCII value of the left most char in a string |
| ATN | Func | Returns the arc tangent of a value expressed in radians |
| AUTO | Cmd | Sets auto line numbering during edit mode |
| BEEP | Func | Produces a 'beep' sound from speaker |
| BLOAD | Cmd | Loads from disk into user specific location in RAM |
| BSAVE | Cmd | Saves specific ranges of RAM onto diskette |
| CALL | Stmt | Transfers control from BASIC to machine code |
| CDBL | Func | Converts value to double precision number |
| CHAIN | Stmt | Loads & runs prg from disk, allows passing of variables |
| CHR$ | Func | Returns the string equivalent of an ASCII value |
| CINT | Func | Rounds values to next whole number |
| CIRCLE | Stmt | To draw an ellipse on the screen |
| CLEAR | Cmd | Sets all variables, strings, and constants to 0, close files |
| CLOSE | Stmt | Close a specific file channel |
| COM(n) | Stmt | Enable/disable trapping of comm. activity |
| COMMON | Stmt | To set-up for passing of variables to chained program |
| CONT | Cmd | Continue program execution after a break encountered |
| COS | Func | Returns the cosine of a value expressed in radians |
| CSNG | Func | To convert a value to a single precision number |
| CSRLIN | Vrbl | Returns the current row position of the cursor |
| CVI | Func | Converts a 2 byte string into its signed decimal equivalent |
| CVS | Func | Converts a 4 byte string into its signed decimal equivalent |
| CVD | Func | Converts a 8 byte string into its signed decimal equivalent |
| DATA | Stmt | Indicator to program that data for READ exists on the line |
| DATE$ | Stmt | Sets the date from a user defined string (MM-DD-YY) |
| DATE$ | Vrbl | Retrieves the current date from string DATE$ |
| DEF FN | Stmt | Defines a user specified function |
| DEF INT | Stmt | To declare variable types as integer |
| DEF SNG | Stmt | To declare variable types as single precision numbers |
| DEF DBL | Stmt | To declare variable types as double precision numbers |
| DEF STR | Stmt | To declare variable types as string of 0-255 chars |
| DEF SEG | Stmt | To define address for BLOAD,BSAVE,CALL,etc |
| DEF USR | Stmt | To specify start address of asm rtn to be called by USR |
| DELETE | Cmd | Deletes specified sections of BASIC |
| DIM | Stmt | Used for setting up dimensioned arrays in memory |
| DRAW | Stmt | Allows drawing of high resolution displays on the screen |
| EDIT | Cmd | Display a specific line from BASIC for editing |
| ELSE | Cmd | Executes when preceding IF statement fails |
| END | Stmt | Ends program execution and returns to OK prompt |
| EOF | Func | Returns a value of (-1) at the end of a disk file in read |
| EQV | | Boolean: x EQV y = 1 if x,y=0,1 or x,y=1,0 else=0 |
| ERASE | Stmt | Eliminates specific dim'd arrays from memory |
| ERR | Vrbl | Returns the error code associated with an error |
| ERL | Vrbl | Returns the error line number associated with an error |
| ERROR | Stmt | To allow simulation of a specific error condition |
| EXIT | Cmd | If SHELL cmd used prior, returns user to BASIC from DOS |
| EXP | Func | To return a value to the power of (n) |
| FIELD | Stmt | To allocate space for variables in a random file buffer |
| FILES | Cmd | Performs a directory of a specific diskette |
| FIX | Func | To truncate a number to a whole number |
| FOR | Stmt | FOR/NEXT: a user defined loop of events to perform |
| FRE | Func | Returns the amount of free RAM in allocated str mem |
| GET | Stmt | To read a record from a random file into a variable buffer |
| GET | Stmt | To transfer graphic images from the screen |
| GOSUB | Stmt | Go to a specific sub-routine in BASIC, with return |
| GOTO | Stmt | Go to a specific section of BASIC code |
| HEX$ | Func | Return the hexadecimal equivalent of an ASCII value |
| IF | Stmt | Question : IF (condition) then perform an operation |
| IMP | | Boolean: x IMP y = 1 if y=1 or x,y=0,0 else=0 |
| INKEY$ | Vrbl | Get a character from the keyboard buffer |
| INP | Func | To return a byte from a specific machine port |

| Command | Type | Description |
|---|---|---|
| INPUT | Stmt | Input a response from the keyboard |
| INPUT# | Stmt | Input a string of characters from diskette |
| INPUT$ | Stmt | To return a string of (n) chars from keyboard buff or file # |
| INSTR | Func | To search for a string within a string, return with position |
| INT | Func | Returns the integer value of a floating point number |
| KEY | Stmt | #, " exp " ;ON;OFF;LIST; - assign f-keys/turn on-off,list |
| KEY(n) | Stmt | To initiate and terminate key capture in program mode |
| KILL | Cmd | Delete a specific file from diskette |
| LEFT$ | Func | Returns a user specified section of string from a string |
| LEN | Func | Returns the length of a string |
| LET | Stmt | Assumed command for assigning variables : Optional Use |
| LINE | Stmt | To draw a high resolution line on the screen |
| LINE INPUT | Stmt | Input a line from keyboard of (1-254) chars no delimiters |
| LINE INPUT# | Stmt | Input a line (254 max) from sequential file, no delimiters |
| LIST | Cmd | Display all or user defined section of BASIC prg |
| LLIST | Cmd | To list all or part of BASIC program in memory to printer |
| LOAD | Cmd | Load a file from diskette into BASIC memory |
| LOC | Func | Returns current position of data in buffer for file access |
| LOCATE | Stmt | Positions and/or turns on cursor anywhere on the screen |
| LOF | Func | Returns number of bytes allocated to a file |
| LOG | Func | Returns the logarithmic equiv. of a number in rads |
| LPOS | Func | Returns current position of line printer print head |
| LPRINT | Stmt | As PRINT; print data at the line printer |
| LPRINT USING | Stmt | As PRINT USING; print data at the line printer |
| LSET | Stmt | Move data from mem to random file buffer, left justified |
| MERGE | Cmd | Merges a BASIC program in ASCII format from disk |
| MID$ | Func | Returns a string from within a string by user specified defs |
| MID$ | Stmt | Replaces a section of a string with a user specified string |
| MKD$ | Func | Converts numeric value to string; double prec expr |
| MKI$ | Func | Converts numeric value to string value; integer expr |
| MKS$ | Func | Converts numeric value to string value; single prec expr |
| MOD | | Modulas arith op: 13 MOD 4=1 (13/4=3, remainder 1) |
| NAME | Cmd | Changes the name of a file on diskette |
| NEW | Cmd | Effectively erases a BASIC program from memory |
| NEXT | Stmt | FOR/NEXT: a user defined loop of events to perform |
| NOT | | Boolean Operand: NOT x = 1 if x=0 else=0 |
| OCT$ | Func | Returns a string of the octal value of a value |
| ON | Stmt | ON (condition) GOTO/GOSUB line#, line#, line#, etc |
| ON COM(n) | Stmt | ON (specific comm condition) GOTO/GOSUB etc. |
| ON ERROR | Stmt | ON (error condition) GOTO/GOSUB etc. |
| ON KEY(n) | Stmt | ON (specific key occurence) GOTO/GOSUB etc. |
| ON PEN(n) | Stmt | ON (specific light pen loc) GOTO/GOSUB etc. |
| ON STRIG(n) | Stmt | ON (specific joy stick cond) GOTO/GOSUB etc. |
| OPEN | Stmt | Open a specific file channel for access |
| OPEN " COM(n) | Stmt | Allocates a RS232 async communications buffer |
| OPTION BASE | Stmt | To declare minimum value for array subscripts |
| OR | | Boolean: x OR y = 1 if x and/or y=1 else=0 |
| OUT | Stmt | To send a byte to a machine output port |
| PAINT | Stmt | To fill in a graphics figure with the selected attribute |
| PEEK | Func | Returns the content of a user defined location in memory |
| PEN | Stmt | ON,OFF,STOP; To read the light pen |
| PEN | Func | To read the numeric value read by the light pen |
| PLAY | Stmt | To play music from string data in program |
| POINT | Func | To read attribute value of a pixel from the screen |
| POKE | Stmt | Stores a user defined value in a user defined loc in RAM |
| POS | Func | Returns the current cursor position on the screen |
| PRINT | Stmt | Print a string of characters to the screen |
| PRINT USING | Stmt | To print strings or numbers with formatting to the screen |
| PRINT# | Stmt | Print a string of characters to an open file |
| PRINT# USING | Stmt | To print strings or numbers with formatting to an open file |
| PSET | Stmt | To display a specific pixel on a high resolution screen |
| PRESET | Stmt | To display a specific pixel on a high resolution screen |
| PUT | Stmt | To write a record from a random file buff to a rnd disk file |
| PUT | Stmt | To transfer graphic images to the screen |

| | | |
|---|---|---|
| RANDOMIZE | Stmt | To re-seed the random number generator |
| READ | Stmt | Read DATA elements from BASIC memory |
| REM | Stmt | Indicator for a comment line in BASIC text |
| RENUM | Cmd | Changes the numbering of a BASIC program in edit mode |
| RESET | Cmd | Close all files and write FAT back to diskette |
| RESTORE | Stmt | Restore all DATA to the start for a READ |
| RESUME | Stmt | Resume program execution after ON ERROR trap |
| RETURN | Stmt | GOSUB/RETURN: return from BASIC sub-routine |
| RIGHT$ | Func | Returns a user specified section of string from a string |
| RND | Func | Returns a random number expressed in decimal notation |
| RSET | Stmt | Move data from mem to random file buffer & right justify it |
| RUN | Cmd | Starts execution of a BASIC program in memory |
| SAVE | Cmd | Saves a BASIC program in memory to diskette |
| SCREEN | Func | To return the value of a specific char on the screen |
| SCREEN | Stmt | To set the screen attributes |
| SGN | Func | Return the sign of a value |
| SHELL | Cmd | Allows entrance into DOS with return from DOS via EXIT |
| SIN | Func | Returns the sine value of a value expressed in radians |
| SOUND | Stmt | To generate sound through the built in speaker |
| SPACE$ | Func | Creates a string of user defined length of ASCII (spaces) |
| SPC | Func | Spaces the cursor over (n) # spaces on the screen |
| SQR | Func | Returns the square root value of a value |
| STICK | Func | To return the x,y co-ordinates of the two joy sticks |
| STOP | Stmt | Stops BASIC execution, returns line # of termination |
| STR$ | Func | Returns the numeric string equivalent of a value |
| STRIG | Func | ON,OFF; to return the status of the joy stick triggers |
| STRIG | Stmt | To read the status of the joy stick triggers |
| STRIG(n) | Stmt | (n) ON,OFF,STOP; to allow use of joystick by trapping |
| STRING$ | Func | Creates a string of user defined length of one ASCII value |
| SWAP | Stmt | Exchanges string variables with each other |
| SYSTEM | Cmd | Pass control of the computer back to DOS |
| TAB | Func | Tabulate the cursor on the screen to a specific position |
| TAN | Func | Returns the tangent of a value expressed in radians |
| TIME$ | Func | To retrieve the current time |
| TIME$ | Stmt | To set the current time (HH:MM:SS) |
| TRON | Cmd | Turn trace of BASIC program on |
| TROFF | Cmd | Turn trace of BASIC program off |
| USR | Func | Pass control of a BASIC prg to asm rtn with return of vars |
| VAL | Func | Returns the numeric value of a string expression |
| VARPTR | Func | To return the address in mem of the vrbl or file ctrl block |
| VARPTR$ | Func | To return addr of 1st byte of data of vrbl before VARPTR |
| WAIT | Stmt | Wait for a certain condition to be met before continuing |
| WEND | Stmt | WHILE/WEND: performs loop till condition is true |
| WHILE | Stmt | WHILE/WEND: performs loop till condition is true |
| WIDTH | Stmt | Set column width of the screen or printer |
| WRITE | Stmt | To output data to the screen in format |
| WRITE# | Stmt | To write data to a sequential file formatted |
| XOR | | Boolean: x XOR y = 1 if x,y=0,1 or x,y=1,0 else=0 |

## The Commodore PC-10 A Brief Look At MS DOS 2.11

MS DOS 2.11, the latest floppy DOS released by Microsoft, is standard with the Commodore PC-10. For those of us who are familiar with Commodores DOS resident in the normal Commodore drives, this is a strange experience. The PC-10 doesn't have intelligent drives, therefore DOS has to be loaded into computer RAM, with disk control being performed by the computers on board processor. Due to this fact the drives tie up computer time to perform all disk activities. Although this is a great loss for fans of normal Commodore drives, this loss is more than made up for by faster disk access via DMA, direct memory access. The drives are dumb, but really quick.

The purpose of this article is to provide a quick run down of the majority of DOS commands available with the standard PC-10. To fully utilize the power of the machine, a working knowledge of DOS is required. And so, the summary is born. Below is a quick reference of most of the commands available. Hope it helps.

| | | | |
|---|---|---|---|
| BREAK | Internal | BREAK ON [d:] BREAK OFF [d:] | Break Off, DOS checks for Break during print or input: Break On, chks always |
| CHKDSK | External | CHKDSK [d]: | Checks the diskette and computer RAM, and reports back with status |
| CLS | Internal | CLS | Clear the display screen |
| COMP | External | COMP filename.ext [d:] filename.ext | Compares files on diskette and reports back differences |
| COPY | Internal | COPY filename.ext [d:] filename.ext [/V] | Copies a specified file onto diskette, the same or different |
| CTTY | Internal | CTTY [Device] | Changes the computer to a remote terminal by re-directing its I/O to Device |
| DATE | Internal | DATE [mm-dd-yy] | Displays current date assignment, and allows user modification |
| DEBUG | External | DEBUG DEBUG filename.ext | High quality machine language monitor for RAM or disk |
| DEL | Internal | DEL filename.ext | Deletes specific files from the directory |
| DIR | Internal | DIR [d:] [/P] [/W] | Performs a passive directory of a diskette to the display screen |
| DISKCOMP | External | DISKCOMP [d:] | Compare two diskettes, and reports differences |
| DISKCOPY | External | DISKCOPY [d:] | Copies the contents of one disk to another, formats as it copies |
| ECHO | Internal | ECHO ON ECHO OFF | Turns the screen Echo of commands in batch file On or Off |
| EDLIN | External | EDLIN | Text editor for creation and manipulation of sequential data files |
| ERASE | Internal | ERASE filename.ext | Delete a specific files from the directory |
| EXE2BIN | External | EXE2BIN filename.ext [d:] [filename.ext] | Converts an .EXE file to a .COM file |
| FORMAT | External | FORMAT [d:] [/S] | Formats a diskette to system compatibility |
| MODE | External | MODE device: specifications | Allows correct set-up for the Line Printer, Serial Port, and Display |
| PAUSE | Internal | PAUSE [remark] | Suspends execution of a batch file till a key is pressed |
| PRINT | External | PRINT filename.ext | Spools data file from disk to printer without affecting computer operation |
| RECOVER | External | RECOVER filename.ext | Recovers and re-creates files as best it can from disk errors |
| REM | Internal | REM [remark] | Flags a comment line in a batch file - displays without action |
| REN | Internal | REN filename.ext [d:] filename.ext | Changes the name of a file on diskette |
| SYS | External | SYS d: | Copies the DOS system files onto a specified diskette |
| TIME | Internal | TIME [hh:mm:ss] | Displays current time assignment, and allows user modification |
| TYPE | Internal | TYPE filename.ext | Prints the contents of a specified file to the screen |
| VER | Internal | VER | Displays the version number and ID of the DOS in use |

# Speeding Up
# Your BASIC Programs

Dr. John W. Ross
Sudbury, Ontario

## Analyze Program CPU Usage . . . And Attack the Slowest Parts!

How would you like to be able to speed up your BASIC programs? Whether you use your computer to print mailing lists, solve systems of partial differential equations, or write the ultimate interactive Star Wars fantasy simulation adventure game, it is a pretty safe bet that you wish your program ran faster. In this article I will show you how to speed up your programs; to do this, we will make use of a special program called a "profiler" to examine the program you want to speed up – but more on this later, first let's take a look at the problem of making a program run faster.

### The 80/20 Rule

The well-known 80/20 rule applies to programs as it does to many situations we encounter from day to day. What it means in terms of program execution is that most programs spend about 80% of the time executing only 20% of the code in the program.

The trick to speeding up your programs is to identify the 20% of the code where the program is spending most of its time, and streamline it as much as possible – you can forget about the rest of the program.

### Code Optimization

Now streamlining a program, or making it run faster, is an art in itself and a complete discussion would easily fill the entire magazine. For our purposes though, there are basically only a couple of ways to make a piece of code run faster – the first is to use a different algorithm, and the second is to use what I call code "tweaking". Modifying the algorithm is the best method if you can do it. For instance, say you have a mailing list program and you have determined that a bubble sort you were using to alphabetize the names was slowing things down. Your best bet would be to use a better sorting algorithm, a Shell sort or Quicksort, say.

Sometimes though, this approach cannot be used, either because there is no better algorithm, or if there is you do not know what it is. In this case we must resort to tweaking; by this I mean the whole set of techniques or "tricks" which make a program run faster – things like not executing REM statements, moving calculations outside FOR–NEXT loops where possible, using variables instead of constants, etc. Often, these techniques are not too well documented, but magazines like Transactor are excellent places to find out about them.

As a final resort, you can take the offending section of code and rewrite it in machine language. If you have done a good job identifying the slowest part of your program, this procedure can lead to really dramatic improvements in execution time. This approach usually requires an intimate knowledge of the computer, and many are reluctant to take it if they do not have to.

The strategy for optimizing code with respect to execution time is quite straightforward, but it requires us to find the parts of our programs which need optimizing. This is the problem; when dealing with even a moderate size program of 100–200 lines, it may be impossible to say for sure where the slowest part or parts are – this is where the profiler comes into play.

### The Profiler

The profiler is a program that runs concurrently with your program and actually measures the amount of time your program spends executing each statement. When your program is finished, the profiler prints out a histogram (an execution time profile) showing the relative amount of time your program spent on each statement – by zeroing in on the histogram peaks, you can easily see where improvements are required. Before going into the profiler design, I would like to discuss an example which shows how it can be used.

### An Example

Some time ago I wrote a 6502 Assembler in BASIC. Although it works very well, it was frustratingly slow. Fig. 1 shows an execution profile of the program produced while it was doing an assembly. Out of 258 lines in the program, only 49 (19%) showed up on the profile; of these 49 "slow" statements, we can see by eye that the program spent most of its time on 9 (18%) of them (the 80/20 rule can often be applied recursively like this).

In fact, we see that there were three bad areas in the program: lines 5–7, lines 12–17 and line 138. The first two locations were part of a parsing routine which scans the input lines – as such they were among the most frequently executed statements in the program. I was able to improve them by some judicious tweaking. The code at line 138 was doing a linear search through a list of opcodes; I was able to improve this part by switching search algorithms to a much faster binary search.

These modifications resulted in a significant improvement in execution speed of the assembler – without the profiler it is safe to say that I could not have made the modifications since I would not have known where they were required. Now let's look at the profiler design.

### Profiler Design

The profiler is written for a CBM 8032 micro, but should be readily adaptable to other CBM models. It is based on the CBM's 60–cycle interrupt; 60 times each second the CBM's 6502 processor runs an interrupt – during this time the video display is updated and the keyboard is scanned. It is quite easy to patch into the interrupt routine. This is an accepted method for running programs concurrently on Commodore computers. What I have done is add some code to examine the storage location which contains the number of the BASIC line currently being executed – a counter for that line is then incremented. Thus, 60 times per second the current line number is sampled and a count maintained for each line; the total of these counts is proportional to the amount of time the program spent executing each line. The counts are displayed in histogram form for a visual indication of the execution profile.

The count for each line is maintained in a 16 bit word and 4k bytes of memory are set aside for counts in the present version of the program.

The profiler is written in two parts – the first part is the interrupt extension which is placed in the CBM's first cassette buffer (starting at memory location $027A) and does the actual profiling – this part is in machine language; the second part is a BASIC program which is loaded after the program to be profiled has executed – this reads the counts and produces the histogram. The assembly listing for the first part is given in Program 1 and the BASIC listing for the second part in Program 2. Program 3 is a loader which loads the machine language program (Program 1) into memory.

**Using the Profiler**

A typical usage pattern would be: (1) load the machine language loader (Program 3) and run it – this resets the top–of–memory pointers and loads the interrupt extension into the first cassette buffer, (2) load the program to be profiled, insert a SYS 634 statement near the beginning, and SYS 658 and SYS 669 statements (these entry points are explained below) as required to profile the appropriate sections of code, (3) run the program to be profiled, (4) load the profile generator program (Program 2) and run it. In more detail, here are the three components of the profiler:

**Profiler Components**

Consider the assembly listing in Program 1. The program has three entry points: SETUP, ACTIVATE and KILL. These are accessed respectively by executing one of . . .

```
SYS 634
SYS 658
SYS 669
```

. . . from the BASIC program to be profiled. These are actually three short subroutines. SETUP initializes the counters in the working storage area to zeroes, ACTIVATE patches in the interrupt extension and KILL removes it. When the extension is patched in, the program segment beginning at MAIN is run automatically 60 times per second. By executing the appropriate subroutine, it is possible to turn the profiler on and off – you may not want to profile your whole program.

The second part of the profiler is the histogram generator shown in Program 2. This is a BASIC program which examines the counts for each line and displays them in a histogram format. The statement which consumed the most execution time is assigned a bar 70 columns wide in the histogram. Other statements are assigned bars whose length is proportional to the amount of execution time they consumed relative to the 70–column statement. If it turns out that a statement's bar would be less than 1 column wide it is not shown.

Look at the listing of Program 2 – it is quite short. The "4" in statement 100 causes the program output to be directed to a printer (it is intended for use with an Epson MX–80). If this is changed to a "3", i.e. "OPEN 1,3", the output will go to the terminal screen instead of the printer.

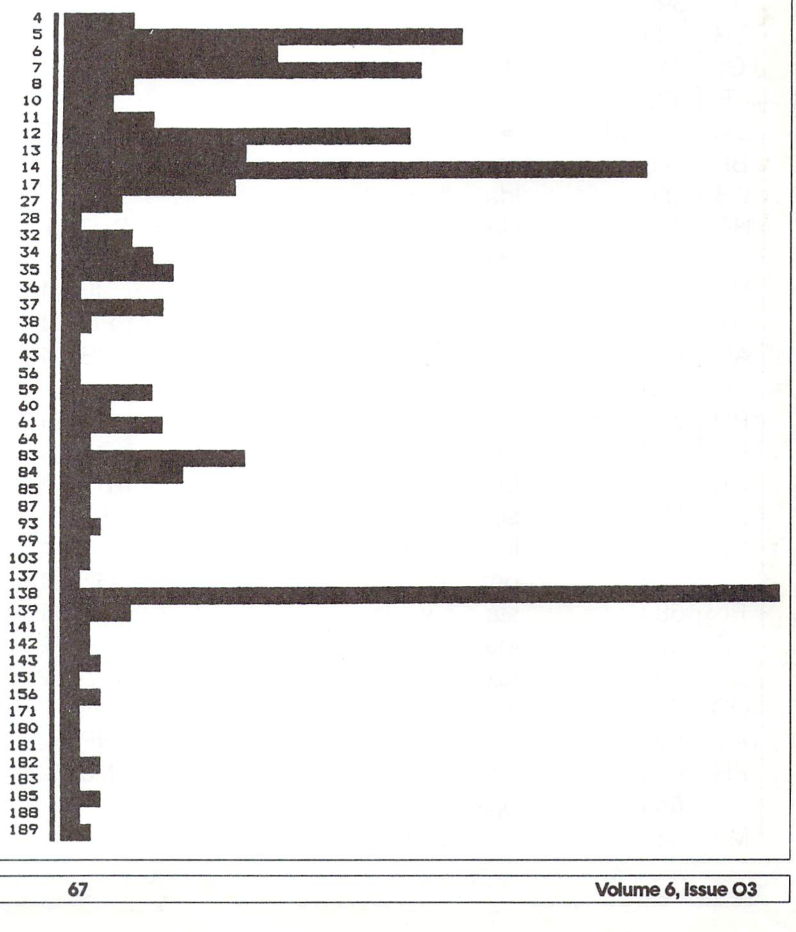
Finally we have the machine language program loader given in Program 3. This is a convenient way to load the program into the computer, and it serves another very important function. Normally, RAM up to hex location $8000 (just below the screen memory) is available for use by BASIC. We require a 4K working storage area and as is usual with a PET, we allocate addresses $7000–$8000 to this purpose. This working storage must be sealed off from BASIC so that it will not be overwritten. This is done by resetting the top–of–memory pointers in line 110 of Program 3. This is a mandatory step before using the profiler.

There is one important restriction to observe about using the profiler – due to its design, programs to be profiled are only allowed to have statement numbers between 1 and 2048 – if the code to be profiled has statement numbers outside this range you will have to use a renumbering utility before running the profiler. In any case you are limited to a 2048 line program, though this should not be a problem – very large programs should be written in modules anyway, and these modules can be profiled individually.

**Summary**

The execution profiler can be one of your most valuable tools in the program design and modification process. So, don't just sit there wondering why your program is taking so long to generate the Klingon invasion force – profile it and see!

**Figure 1:** Sample Profile Of A BASIC Program (1/2 actual size)



**Program 1:** 6502 Assembly Language Portion Of The Interrupt-Driven Profiler

```
PJ  100 *          =    $027a
EO  110 ;
NA  120 zw         =    $00
EA  130 iv         =    $90
HD  140 cline      =    $36
LK  150 intrpt     =    $e455
GB  160 ;
KC  170 setup      =    *
MO  180            lda  #$70
OC  190            sta  zw+1
DL  200            lda  #0
AI  210            sta  zw
LK  220            tay
MF  230 ;
JG  240 loop       =    *
DA  250            sta  (zw),y
EJ  260            inc  zw
HK  270            bne  loop
KG  280            inc  zw+1
AL  290            ldx  zw+1
FN  300            cpx  #$80
PM  310            bne  loop
MC  320            rts
AM  330 ;
EK  340 activate   =    *
HP  350            sei
JC  360            lda  #<main
NA  370            sta  iv
JD  380            lda  #>main
DO  390            sta  iv+1
OB  400            cli
GI  410            rts
KB  420 ;
DA  430 kill       =    *
BF  440            sei
CB  450            lda  #<intrpt
HG  460            sta  iv
CC  470            lda  #>intrpt
ND  480            sta  iv+1
IH  490            cli
AO  500            rts
EH  510 ;
PD  520 main       =    *
NF  530            ldy  #0
HL  540            lda  cline
EN  550            sta  zw
CL  560            lda  cline+1
LA  570            ora  #$70
EL  580            sta  zw+1
JB  590            lda  (zw),y
DC  600            tax
OD  610            inx
AC  620            txa
PH  630            sta  (zw),y
HP  640            bne  out
MA  650            lda  cline+1
NH  660            ora  #$78
OA  670            sta  zw+1
DH  680            lda  (zw),y
NH  690            tax
IJ  700            inx
KH  710            txa
JN  720            sta  (zw),y
AF  730 ;
LN  740 out        =    *
CE  750            jmp  intrpt
```

**Program 2:** BASIC Program To Display Profiler Results

```
LO  100 rem profiler - basic portion
DC  110 open 1,4
EL  120 dim tt%(2048,2): j=0: tm=0
LG  130 d=8*256: lo=7*4096: hi=lo+d
KJ  140 for i=lo to hi-1: sn=i-lo
KB  150 t=peek(i+d)*256+peek(i)
PJ  160 if t=0 then 190
HO  170 j=j+1: tt%(j,1)=sn: tt%(j,2)=t
KG  180 if t>tm then tm=t
CM  190 next
EM  200 for i=1 to j
FJ  210 a$=chr$(181)
MG  220 pc=int(tt%(i,2)/tm*70)
NF  230 if pc=0 then 260
BL  240 for j=1 to pc: a$=a$+chr$(223): next
HN  250 print#1,right$("[3 spaces]"
          +str$(tt%(i,1)),4)"[1 space]";a$
IA  260 next
LL  270 print#1: close1
IB  280 end
```

**Program 3:** BASIC Program To Load Profiler

```
JI  100 rem profiler loader
PC  110 poke 52,0: poke 53,112: clr
OB  120 read n,l: for i=1 to n: read x: poke l,x: l=l+1
          : next: end
NP  130 data  83, 634
GJ  140 data 169, 112, 133,   1, 169,   0, 133,   0
BJ  150 data 168, 145,   0, 230,   0, 208, 250, 230
PI  160 data   1, 166,   1, 224, 128, 208, 242,  96
FN  170 data 120, 169, 168, 133, 144, 169,   2, 133
GF  180 data 145,  88,  96, 120, 169,  85, 133, 144
NA  190 data 169, 228, 133, 145,  88,  96, 160,   0
FA  200 data 165,  54, 133,   0, 165,  55,   9, 112
LO  210 data 133,   1, 177,   0, 170, 232, 138, 145
AN  220 data   0, 208,  13, 165,  55,   9, 120, 133
NP  230 data   1, 177,   0, 170, 232, 138, 145,   0
MJ  240 data  76,  85, 228
```

# Hi-Res Text Maker

Darren James Spruyt
Gravenhurst, Ontario

## Scaled Text For Your Hi-Res Screen!

This program allows one to reproduce any of the C-64 characters on the hi-res screen with its X dimension enlarged up to X25 and the Y dimension up to X40. This is useful for any program that needs a slightly larger text size.

The program is very easy to use. The following is a list of the parameters needed by the routine and where to poke the needed values.

| POKE | USE |
|---|---|
| 678 | X co-ordinate (0–39) |
| 679 | Y co-ordinate (0–24) |
| 681 | X multiple for size (1–40) |
| 682 | Y multiple for size (1–25) |
| 683 | char number (poke values) |
| 820 | overwrite (1 = yes/0 = no) |
| 821 | color byte |

The overwrite allows character to be put on top of each other and 'mesh' together rather than having the area erased before a new character is put on. The color bytes upper 4 bits or nybble are for the character color while the lower four bits specify the character's background color.

There are also some enabling SYS's:

SYS 32768 makes the hi-res screen visible;
SYS 32771 clears the hi-res screen and fills color memory with the background color;
SYS 32774 reverts back to the original text screen, on which no changes have been made;
SYS 32777 plots the character.

The program uses memory from $8000–81C5 for the program and from $5C00–$8000 for the hi-res screen and color map.

To protect the hi-res screen and the color map from being overwritten, set the limit of memory with:

POKE 55,0 : POKE 56,92

Listing 1 is a short demonstration of the hi-res text program, listing 2 is the BASIC loader, and finally listing 3 is the PAL source code.

**Listing 1: BASIC Demo Program**

```
GA  100 rem sample program to use hi-res
MJ  110 rem          text
LF  120 rem darren james spruyt 85/06/01
AK  130 rem
JM  140 poke 53281,0 :rem set bg color
BP  150 sys 32771 :rem clear hi-res screen
GD  160 sys 32768 :rem turn screen on
HG  170 rem 820 is overlap reg
HN  180 rem 821 is color reg
CD  190 rem 681 is xsize reg
BE  200 rem 682 is ysize reg
IJ  210 rem 679 is y pos reg
NJ  220 rem 678 is x pos reg
LC  230 rem 683 is char  reg
LH  240 for k = 1 to 15
II  250 a$ = "the"
JF  260 poke 821,k*16
NN  270 poke 681,2
IO  280 poke 682,2
AD  290 for j = 1 to len(a$)
CA  300 poke 679,3
OC  310 poke 678,2 + ((j-1)*3)
NF  320 poke 683,asc(mid$(a$,j,1))
BD  330 sys 32777
GD  340 next j
PD  350 poke 681,8
NE  360 poke 682,9
HM  370 poke 821,k*16
MA  380 poke 679,10
FF  390 poke 678,1
BJ  400 poke 683,asc("t")
BI  410 sys 32777
ID  420 for n = 1 to 15
MG  430 a$ = "ransactor"
KI  440 poke 681,3
CJ  450 poke 682,2
KN  460 for j = 1 to len(a$)
CJ  470 c = n + (j-1):if c>15 then c = c-15
NB  480 poke 821,c*16
KI  490 poke 679,14
LP  500 poke 678,7 + ((j-1)*3)
LB  510 poke 683,asc(mid$(a$,j,1))
PO  520 sys 32777
EP  530 next j
KA  540 next n
LA  550 next k
OA  560 sys 32774
```

**Listing 2: BASIC Loader**

```
DD 100 print"Sqqh-res text maker by
JF 110 print"darren james spruyt
BM 120 print"as of june1/85
PG 130 rem start of basic loader code
DH 140 read a,b,d
GH 150 print"qnow loading in code."
KJ 160 for k=a to b
GC 170 read c:poke k,c
DH 180 poke 1024,c:poke55296,c
PP 190 ch=ch+c:next
GK 200 if ch<>d then print"data error":stop
OK 210 print"qdone.":end
CH 220 data  32768 , 33223 , 47644
FD 230 data  76,  95, 129,  76, 143, 129,  76, 119
IF 240 data 129, 169,  96, 133,  35, 169,   0, 174
DN 250 data 167,   2, 240,  12,  24, 105,  64, 144
BO 260 data   2, 230,  35, 230,  35, 202, 208, 244
JB 270 data 174, 166,   2, 240,  10,  24, 105,   8
BG 280 data 144,   2, 230,  35, 202, 208, 246, 133
EF 290 data  34, 120, 165,   1,  41, 251, 133,   1
HO 300 data 169,   0, 133,  21, 173, 171,   2, 133
IG 310 data  20,   6,  20,  38,  21,   6,  20,  38
LM 320 data  21,   6,  20,  38,  21,  24, 169, 216
MA 330 data 101,  21, 133,  21, 160,   7, 177,  20
DD 340 data 153, 174,   2, 136,  16, 248, 165,   1
CE 350 data   9,   4, 133,   1,  88, 169,   0, 141
LI 360 data 187,   2, 141, 173,   2, 173, 170,   2
BJ 370 data 141, 172,   2, 173, 173,   2, 141, 185
NJ 380 data   2, 174, 187,   2, 160,   7, 126, 174
BA 390 data   2, 176,   3, 169,   0,  44, 169, 255
DL 400 data 153,  60,   3, 136,  16, 240, 126, 174
FM 410 data   2, 169,   0, 141, 182,   2, 169,   8
DN 420 data 141, 186,   2, 162,   0, 172, 169,   2
IF 430 data 189,  60,   3, 240,   7,  56,  46, 182
GF 440 data   2,  76, 176, 128,  24,  46, 182,   2
GB 450 data 206, 186,   2, 240,  11, 136, 208, 232
LD 460 data 232, 224,   8, 208, 224,  76, 241, 128
CA 470 data 140, 183,   2, 142, 184,   2, 172, 185
KO 480 data   2, 173,  52,   3, 240,   2, 177,  34
HA 490 data  13, 182,   2, 145,  34, 169,   0, 141
AD 500 data 182,   2, 169,   8, 141, 186,   2, 173
CL 510 data 185,   2,  24, 105,   8, 141, 185,   2
EC 520 data 172, 183,   2, 174, 184,   2,  76, 181
GB 530 data 128, 238, 173,   2, 173, 173,   2, 201
GL 540 data   8, 208,  18, 169,   0, 141, 173,   2
LA 550 data  24, 165,  34, 105,  64, 144,   2, 230
PP 560 data  35, 230,  35, 133,  34, 206, 172,   2
AB 570 data 240,   3,  76, 115, 128, 238, 187,   2
LM 580 data 173, 187,   2, 201,   8, 240,   3,  76
AI 590 data 109, 128, 169,  92, 133,  21, 169,   0
IB 600 data 174, 167,   2, 240,  10,  24, 105,  40
BE 610 data 144,   2, 230,  21, 202, 208, 246,  24
AC 620 data 109, 166,   2, 133,  20, 144,   2, 230
KH 630 data  21, 174, 170,   2, 172, 169,   2, 136
CI 640 data 173,  53,   3, 145,  20, 136,  16, 251
HE 650 data 165,  20,  24, 105,  40, 144,   2, 230
LN 660 data  21, 133,  20, 202, 208, 230,  96, 173
KP 670 data  17, 208,   9,  32, 141,  17, 208, 169
DN 680 data 120, 141,  24, 208, 173,   0, 221,  41
AC 690 data 252,   9,   2, 141,   0, 221,  96, 173
OD 700 data  17, 208,  41, 223, 141,  17, 208, 169
BJ 710 data  21, 141,  24, 208, 173,   0, 221,  41
ID 720 data 252,   9,   3, 141,   0, 221,  96, 160
KK 730 data   0, 169,  96, 133,  21, 132,  20, 162
DB 740 data  32, 169,   0, 145,  20, 136, 208, 251
ID 750 data 230,  21, 202, 208, 246, 173,  33, 208
PM 760 data  41,  15, 133,   2,  10,  10,  10,  10
GO 770 data 160,   0,   5,   2, 153,   0,  92, 153
FG 780 data   0,  93, 153,   0,  94, 153, 232,  94
CP 790 data 200, 208, 241,  96,   0,   0,   0,   0
```

**Listing 3: PAL Source Code**

```
GL 100 rem hi-res text maker
EE 110 rem by darren james spruyt
GO 120 rem box 1226
FP 130 rem gravenhurst, ontario
EJ 140 rem p0c 1g0
EL 150 rem
BH 160 sys 700
CB 170 .opt oo
KJ 180 *=$8000
OF 190 base   =   $6000
LK 200 temp   =   $02b6
NL 210 tmp1   =   $02b7
LM 220 tmp2   =   $02b8
IF 230 pntr1  =   $02b9
IG 240 cntr1  =   $02ba
NH 250 charow =   $02bb
DI 260 cntr2  =   $02ac
DJ 270 cntr3  =   $02ad
DD 280 color  =   $0335
BD 290 additi =   $0334
CK 300 ;
HH 310 ;followin jmp table
II 320        jmp  hion
FA 330        jmp  clear
ED 340        jmp  hioff
EN 350 ;
CF 360 ;start of code
KG 370 print  =    *
CP 380 ;
GH 390 ;create base address
EC 400        lda  #>base        ;set high address
IM 410        sta  $23           ;set high address
LP 420        lda  #<base        ;a=lo address
```

```
DP  430       ldx   $02a7        ;character row
GG  440       beq   p1           ;for each
PM  450  p3   clc                ;row
CA  460       adc   #$40         ;add to base
JE  470       bcc   p2           ;address, 320
CE  480       inc   $23          ;or $0140 in hex
FP  490  p2   inc   $23          ;
FD  500       dex                ;done
OG  510       bne   p3           ;no
IP  520  p1   ldx   $02a6        ;column address
IM  530       beq   p4           ;for each column
FG  540  p6   clc                ;add 8
HB  550       adc   #$08         ;to the base
HM  560       bcc   p5           ;address
PM  570       inc   $23          ;
PC  580  p5   dex                ;doneprint
EM  590       bne   p6           ;no
ME  600  p4   sta   $22          ;save lo address
IN  610  ;
AG  620  ;copy char data from rom
DN  630  ;to $02ae
NP  640       sei                ;lockout irq
JA  650       lda   $01
BN  660       and   #%11111011;make d rom
FP  670       sta   $01          ;visible
MN  680       lda   #$00         ;generate
NH  690       sta   $15
IN  700       lda   $02ab        ;indirect
OI  710       sta   $14
OJ  720       asl   $14          ;based
AM  730       rol   $15
OC  740       asl   $14          ;on
EN  750       rol   $15
JE  760       asl   $14          ;character number
IO  770       rol   $15
HA  780       clc                ;times 8
FJ  790       lda   #$d8
KO  800       adc   $15          ;plus $d000
FP  810       sta   $15
CF  820       ldy   #$07         ;copy character
EI  830  l1   lda   ($14),y
AJ  840       sta   $02ae,y      ;bit patterns
NA  850       dey
PH  860       bpl   l1           ;from rom
FO  870       lda   $01
HA  880       ora   #%00000100
NH  890       sta   $01          ;close rom up
EK  900       cli                ;release irq
PK  910       lda   #0           ;initialize
PA  920       sta   charow       ;char pixel rows
DD  930       sta   cntr3        ;screen pixel row
JE  940  z15  lda   $02aa
CE  950       sta   cntr2        ;y multiple size
OC  960  z13  lda   cntr3
HF  970       sta   pntr1        ;y val for screen
AI  980       ldx   charow       ;current char row
BA  990       ldy   #$07         ;break bits
FB  1000 z1   ror   $02ae,x      ;into bytes at
FM  1010      bcs   z2           ;$033c
PB  1020      lda   #$00
BO  1030      .byte $2c
LB  1040 z2   lda   #$ff
PE  1050      sta   $033c,y
NG  1060      dey                ;done break
GM  1070      bpl   z1           ;no
JM  1080      ror   $02ae,x      ;finish rotation
NC  1090      lda   #0
MP  1100      sta   temp         ;set temp
JJ  1110      lda   #$08
NN  1120      sta   cntr1        ;set rotations 8
JO  1130      ldx   #$00
ON  1140 z8   ldy   $02a9        ;get x multiple
MA  1150 z5   lda   $033c,x      ;test bit values
PC  1160      beq   z3           ;zero means 0
IJ  1170      sec                ;rotate a 1 in
CG  1180      rol   temp
DC  1190      jmp   z4
HK  1200 z3   clc                ;rotate a 0 in
AI  1210      rol   temp
DK  1220 z4   dec   cntr1        ;done 8 shifts
AI  1230      beq   z6           ;yes – to screen
LB  1240 z7   dey                ;check multiples
GP  1250      bne   z5           ;do more
BO  1260      inx                ;check all 8 bits
DK  1270      cpx   #$08         ;are done
OF  1280      bne   z8           ;no – do more
CE  1290      jmp   z9           ;
EF  1300 z6   =     *            ;
ED  1310      sty   tmp1         ;save y
LD  1320      stx   tmp2         ;save x
IE  1330      ldy   pntr1        ;get y pntr
FM  1340      lda   additi       ;mesh mode
MA  1350      beq   z23          ;no
OF  1360      lda   ($22),y      ;get prev pattern
NC  1370 z23  ora   temp         ;add new pattern
NB  1380      sta   ($22),y      ;back to screen
KG  1390      lda   #0           ;set temp to zero
DC  1400      sta   temp
AG  1410      lda   #8           ;set cntr
GI  1420      sta   cntr1
JP  1430      lda   pntr1        ;add 8 tp pntr1
MH  1440      clc                ;to get
HM  1450      adc   #$08         ;to the next
KL  1460      sta   pntr1        ;row
IJ  1470      ldy   tmp1         ;restore x and y
CD  1480      ldx   tmp2
ND  1490      jmp   z7           ;recurse
EA  1500 z9   inc   cntr3        ;count pixel rows
EK  1510      lda   cntr3
EE  1520      cmp   #$08         ;at eight
EK  1530      bne   z10          ;nope
HC  1540      lda   #$00
```

```
GO  1550        sta   cntr3        ;re-set counter
OI  1560        clc
HI  1570        lda   $22          ;add 320
NE  1580        adc   #$40
MM  1590        bcc   z11          ;to the indirect
JO  1600        inc   $23
IM  1610 z11    inc   $23          ;address ($22)
IB  1620        sta   $22
CM  1630 z10    dec   cntr2        ;y multiples
CH  1640        beq   z12          ;done
LF  1650        jmp   z13          ;repeat previos ro
KI  1660 z12    inc   charow       ;chr  pixel row
KH  1670        lda   charow       ;
AI  1680        cmp   #$08         ;done all 8 rows
FJ  1690        beq   z14          ;yes then finished
DF  1700        jmp   z15          ;do next row
PM  1710 z14    =     *
JG  1720 ;add colour as indicated
HF  1730        lda   #$5c         ;build the
HJ  1740        sta   $15
MN  1750        lda   #$00         ;indirect
JP  1760        ldx   $02a7
DL  1770        beq   j2           ;address
PP  1780 j3     clc
AN  1790        adc   #$28         ;via base
NA  1800        bcc   j1
JP  1810        inc   $15          ;of $5c00
IE  1820 j1     dex
JA  1830        bne   j3           ;plus y pos *40
ID  1840 j2     clc
EI  1850        adc   $02a6        ;and  x pos
MA  1860        sta   $14
PF  1870        bcc   j7
GC  1880        inc   $15          ;done
PA  1890 j7     =     *
DF  1900        ldx   $02aa        ;get y size
LJ  1910 j6     ldy   $02a9        ;get x size
LD  1920        dey
EP  1930        lda   color        ;get color val
IK  1940 j4     sta   ($14),y      ;put in mem
LD  1950        dey                ;done x
GD  1960        bpl   j4           ;no
NL  1970        lda   $14          ;add 40
CD  1980        clc
EG  1990        adc   #$28         ;to the address
NN  2000        bcc   j5
IK  2010        inc   $15          ;done
NJ  2020 j5     sta   $14
II  2030        dex                ;done y
IG  2040        bne   j6           ;no
OO  2050        rts
FM  2060 hion   =     *
PK  2070        lda   $d011
CK  2080        ora   #%00100000;turn hi-res bit
BA  2090        sta   $d011
DN  2100        lda   #%01111000
GN  2110        sta   $d018        ;set screen/map
IB  2120        lda   $dd00
KH  2130        and   #%11111100
EP  2140        ora   #%00000010
MM  2150        sta   $dd00        ;set vic chip
BA  2160        rts                ;addresses
JL  2170 hioff  =     *
NB  2180        lda   $d011
BC  2190        and   #%11011111;re-set bit map
PG  2200        sta   $d011
EC  2210        lda   #21
EO  2220        sta   $d018        ;reset screenmap
GI  2230        lda   $dd00
IO  2240        and   #%11111100
DG  2250        ora   #%00000011
ED  2260        sta   $dd00        ;reset vic chip
EI  2270        rts                ;address
OG  2280 clear  ldy   #0
IM  2290        lda   #>base       ;base address
GA  2300        sta   $15          ;into
GB  2310        sty   $14          ;($14)
AD  2320        ldx   #32          ;do 32 pages
BI  2330        lda   #00          ;
AP  2340 t6     sta   ($14),y      ;zero memory
JO  2350        dey
KH  2360        bne   t6
PO  2370        inc   $15
GC  2380        dex                ;doneprint
AN  2390        bne   t6           ;no
PE  2400        lda   $d021        ;pull old color
GF  2410        and   #%00001111;from vic chip
ED  2420        sta   $02
JC  2430        asl
DD  2440        asl
BJ  2450        asl                ;shift to high
KN  2460        asl                ;nybble
BP  2470        ldy   #0
BO  2480        ora   $02          ;low nybble
OH  2490 j53    sta   $5c00,y      ;fill
II  2500        sta   $5d00,y      ;color
IM  2510        sta   $5e00,y      ;area
NL  2520        sta   $5f00-24,y   ;up
CM  2530        iny
LL  2540        bne   j53
CO  2550        rts
MN  2560 .end
```

# The SAVE@ Debate Rages On – A Few More Observations

### SAVE@ Gap Attack!

Finally, that small ulcer that was acting up every time I used SAVE@ has started to heal. Thanks to Charles Whittern for demonstrating that the BUG really exists.

I used the SAVE@ EXPOSED!!! program with a slight modification so that every time the directory is checked and the names of the program pairs SAVED@ is printed on the printer along with the program–start track and sector. Also, the routine checks for any programs that start with the same track/sector (the clone phenomenon). When such a situation is detected, the program prints the two filenames and their track/sector pointers. So one can just RUN the program and do something else. Checking after 15 to 20 minutes would indicate that SAVE@ has done its thing!

I found that disks which have 'holes' in the directory are especially sensitive to SAVE@. I used such a disk and after about 7 RUNs there was a corrupted file. Then I scratched one of the clones, validated the disk and repeated the above once again with the same results. Then I ran DIRECTORY GAP REMOVER (Richard Evers, Transactor 5(6): 57, 1985). Running SAVE@ EXPOSED!!! required 34 RUNs before a file was corrupted again. I think directory gaps contribute somehow to the susceptibility of a disk to SAVE@–induced damage.

I also found that to further guard against SAVE@, one should bring the file on which one is working (and which will be SAVED@) to the end of the directory. What I do is LOAD the file after RUNning GAP FILL, then SAVE it as "TEMP". Then I work with this file till I get it right using SAVE@. At this point I scratch the original file and SAVE TEMP with the right filename. I know this is tedious but I consider it much better than loading SPEEDSCRIPT and finding that it is actually PIANO64 in disguise!

Ranjan Bose, Winnipeg, Manitoba

### What We Have Here Is A Failure To Re–Allocate

Charles Whittern's July article on the 1541 SAVE@ bug will no doubt elicit a flurry of activity on that long rumoured but previously unconfirmed gremlin. A simple manifestation of the bug can be demonstrated as follows: LOAD a ten block BASIC program file and SAVE it four times (under different filenames) to a newly NEWed disk. LOAD/LIST the directory to confirm that 624 blocks are free. LOAD the program and SAVE@ the fourth then the third file. Initialize the drive (or cold start your C–64) to get rid of the previous BAM then LOAD/LIST the directory again. Surprise! 634 BLOCKS FREE! A look at the BAM and file chains reveals that sectors used by the third file's replacement are not allocated in the BAM. That is, the original sectors occupied by file three are de–allocated normally but the newly occupied sectors do not get allocated. And there sits file three, accessible and functional but just waiting for a subsequent write to wander into its unprotected space. Why some SAVE@'s work OK and others do not is no doubt a crucial question. It is now clear that the SAVE@ bug results from a failure to allocate.

Phil McBrayer, Lexington, KY

**Editors Note:** *My 1541 seems to be immune to this problem. It may be a problem that is dependent on ROM revision.*

### The Relentless SAVE@

Accolades to you and Charles Whittern for your definitive work with "SAVE With Replace Exposed!!".

I would like to mention two associated thoughts or suggestions or questions, however they may be taken:

1. The first time Save@ bit me, about a year ago when I had had my 64 for four or five months, the names of a program about 30 blocks and a program of about 8 blocks interchanged. My point is that as I remember, and it was quite a while ago, there was no way that I could scratch the two programs and put them back in right with plain "save". They insisted on being reversed. I ended up putting programs I wanted to keep on a new disk and re-formatted the old disk. If that is true, it ought to be some sort of a hint of what gets mixed up.

2. When I bought my 64, I bought Easy Script, which I have used heavily and love more than you would ever believe. Praying that what I am about to say doesn't bring the roof down on me (I am "knocking on wood" madly), Easy Script has never loused with replace for me, and I have used it far, far more than I have used Save@ with plain Basic programs. Of course, Easy Script is machine language, protected, and for all I know it may Scratch before Saving. It wouldn't be hard to manually Scratch before Saving because Easy Script has a slick disk mode which doesn't affect the text in memory, but I just haven't as yet found it necessary.

But beyond Easy Script, I have a program which I originated, in Basic, which I have updated 28 files weekly for 32 weeks, now, using Save@ from within the program. Again knocking on wood, these updates haven't as yet messed up. On the other hand, there is never much change in length of the files and they only occupy two blocks each. However, I happened to look at the directory the other day and the disk showed only 40 blocks free. I ran the "validate" command which increased free blocks to 584; I am hoping that this will not trigger a Save@ problem.

From these two cases I had a theory that Save@ works perfectly from within a program, but Charles Whittern's experiment rather blows that. Now I am wondering if sequential files, which both of my illustrations are, may be immune to the problem.

At any rate, I hope that you experts and Commodore continue your research until all ramifications of the problem are known.

H.C. Doennecke, Tulsa, OK

**Editors Note:** *Who knows, sequential files might be immune. Program files only use one data buffer within the drive during creation, sequential files consume two. It could be that Commodore drives are claustrophobic, therefore flying into spastic rages whenever confronted with the evil Save@.*

### SAVE@ Traps & Tips

If you insert Validate into the LOAD–SAVE@ Whittern loop, there's no longer any file damage.

You can also intentionally damage files by (a) LOADing a program, (b) SCRATCHing it and several other programs, then (c) SAVEing the program back to disk. Again, if you Validate the disk after the SCRATCHes, before the SAVE, the DOS error is prevented.

Finally, here's a good way to produce highly unreliable disks which will either not work, crash within a few days, or give occasional unexplained file errors: (1) Buy the cheapest bulk disks. (2) Don't reset the disk drive before you format them. And, (3) use a faster than normal method to format them. The fast disk copier programs or speeded-up 1541 ROMs are particularly handy for this purpose.

John R. Menke, Mt. Vernon, IL

**SAVE@ Goes One Degree Too Far**

I read Charles H. Whittern's article 'SAVE With Replace Exposed!!' in the Transactor. I consider this a very serious situation.

Recently, a large part of my Master's Thesis was destroyed by a word processor I was using on the C-64, jeopardizing my degree! When I wrote the software vendor, they shrugged off the problem with a form letter blaming the SAVE. I'm not sure where the responsibility lies, but I feel that Commodore and the software vendors have a responsibility to provide immediate relief. If they do not take this matter seriously, a law suit would be in order.

Can you help me contact Charles Whittern and anyone else that is resolving this problem?

Daniel Bresnahan, Bloomfield, New Jersey

**The Instigator Returns!!!**

Thank you for publishing my research on the Save@ phenomena. If you have not yet sent the champagne, I would be happy to accept a copy of the new "Complete Commodore Inner Space Anthology" in its place. Perhaps this would be easier for you to ship, and it would be of much more use to me as a non-drinker (Although I was going to keep the bottle as a trophy!).

I modified my "SAVE@ EXPOSED!!!" program recently to include a VERIFY of each program immediately after it is SAVED@. Also I added a POKE 198,0 to HALT the program if a VERIFY error occurred. This is skipped over if the VERIFY is ok. I figured this would catch the first incorrect replacement and HALT the program. After RUNning this version for a while, I was amazed to find that although each program LOADED, SAVED@, and VERIFIED ok, SAVE@ was still up to its old tricks! After each SAVE@ the VERIFY showed that the program just placed on disk matched byte for byte the one placed in memory. Yet LOADing and LISTing the programs revealed several of them to be very different indeed! How can this be? Now I am truly baffled! I am sending this program along in hopes that it will aid in finding the source of the trouble.

Another thing that I have discovered is that the BAM gradually fills as "SAVE@ EXPOSED!!!" RUNs, until it is completely allocated. The block counts do not reflect this increase although the blocks free does (it takes several hours to accomplish this).

Charles H. Whittern, President
Lenawee Users Group – Commodore 64 (LUG-64)
Hudson, Michigan

```
BE  100 rem "save@ & verify"
JO  110 rem may 14, 1985 by c.h. whittern, box 215,
        hudson, mich  49247
ON  120 cs$ = chr$(147): qt$ = chr$(34)
GL  130 d1$ = chr$(17): d2$ = d1$ + d1$: d3$ = d2$ +
        d1$ : d4$ = d3$ + d1$: d5$ = d4$ + d1$
HJ  140 for i = 1 to 5: read a$(i): next
IL  150 i = int(rnd(0)*5) + 1
JD  160 print cs$"load"qt$;a$(i);qt$",8"
NH  170 print d4$"save"qt$"@0:"a$(i);qt$",8"
NO  180 print d3$"verify"qt$;a$(i);qt$",8"
IB  190 print d5$"poke 198,0"
EN  200 print d1$"load"qt$"save@ + verify"qt$",8"
AP  210 poke 631,19: for i = 1 to 5: poke 631 + i,13
        : next: poke 637,82
EC  220 poke 198,9: end
PD  230 data recover ram,check disk drive,quadra,
        performance test,disk log
```

**Editors Note:** *The following is an excerpt from a letter recently sent to us by Ray Quiring. We originally received a letter from Mr. Quiring back in September of 1984 stating that he had finally found the SAVE@ bug. At that time, we could not reproduce the bug using the information he supplied. His bug reproduction technique was to create a disk error then SAVE@ a file while the error was still present. We tried, but the drive we were using worked just fine. With that background supplied, the following letter should make a bit more sense.*

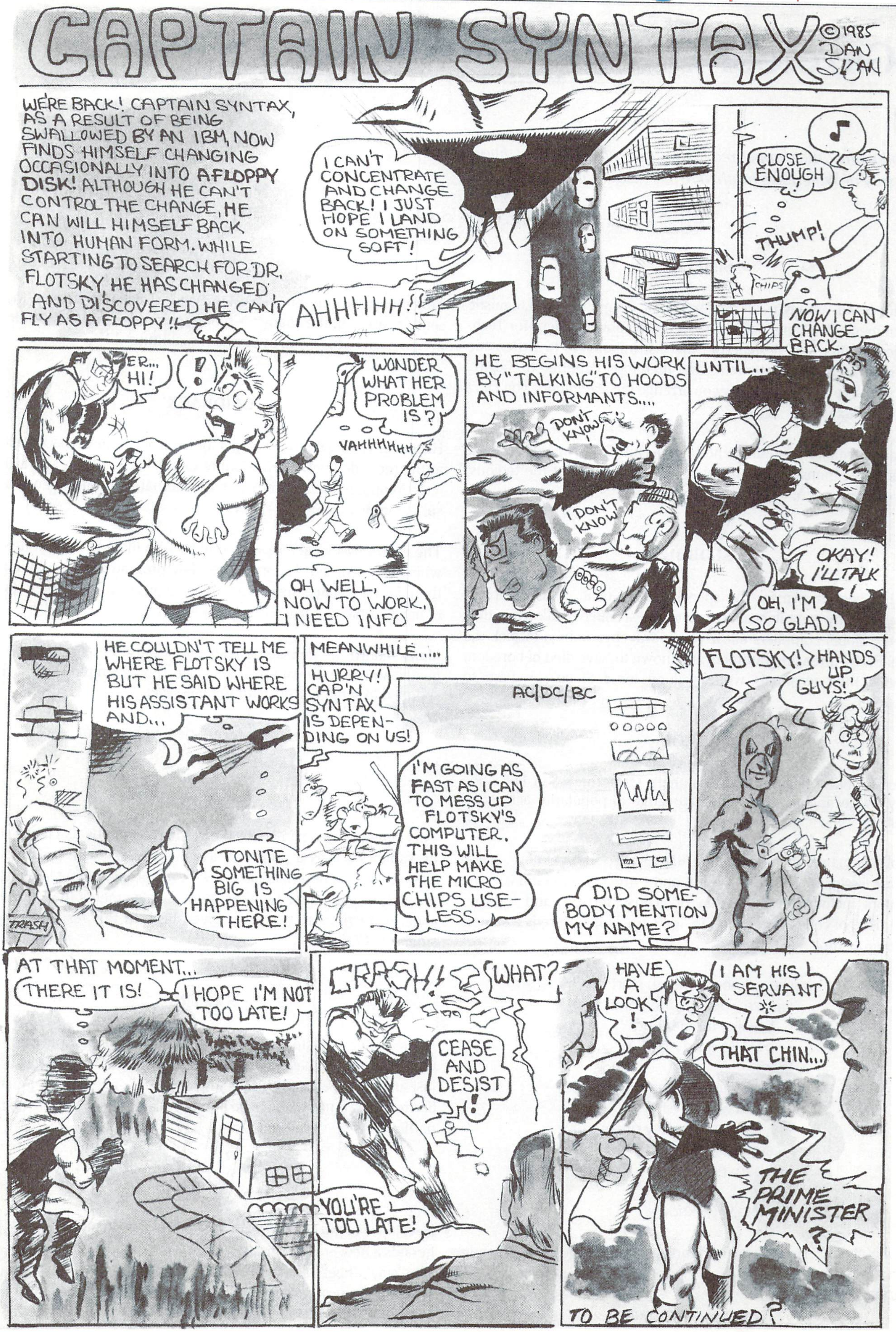
**The Disappearing SAVE@**

The circumstances surrounding the disappearance of the bug gives another clue as to what is happening. The procedure worked perfectly on both my drives, that is it would cause two files to point to the same track and sector. But then the drive misalignment became severe and both were eventually sent out to be realigned. When they came back the bug was nowhere to be found. This only reinforces my belief that the bug appears as a response to some DOS error condition. We never notice most DOS errors because the DOS tries several times before giving up and reporting the error.

The explanation of the symptoms of the SAVE@ bug is straight forward: nothing can ever be correct after two files point to the same track and sector. If the sector happens to be de-allocated in the BAM, the very next SAVE will try to use the sector as if it were available. Mysteriously, the two old files will point to part or all of the new file saved. How much of the new file gets linked in depends upon how much of the new file was saved before the DOS used the sector which, unknown to the DOS, was already "in use" by the two previous files. You never find out about the problem until you try to use one of the two previous files. Detecting the multiple use of the same sector is too much to ask of a DOS, so what should have been done to prevent it? One thing that would have helped is to issue an error in the attempt to de-allocate a sector that is already de-allocated. The DOS does not presently do this. This would, at least, have flagged the condition early and may even have prevented the damage in the first place. It is understandable why the designers of the DOS did not do this: why prepare for a condition that logically should never occur?

All the other symptoms of the SAVE@ bug are explained by analysing the various combinations possible of two or more files pointing to the same sector, and the sector being allocated or de-allocated at any given time. This does not explain where the bug originates. I believe that the bug can be used as a sensitive test of drive condition. When the drive is in good shape the bug stays hidden, when the drive suffers from heat prostration or head misalignment the bug reappears. Prevention of the bug by resetting prior to and after using SAVE@ may not be as sure a thing as it has been for me.

Ray Quiring, Kerby, Oregon


CAPTAIN SYNTAX
©1985 DAN
WE'RE BACK! CAPTAIN SYNTAX, AS A RESULT OF BEING SWALLOWED BY AN IBM, NOW FINDS HIMSELF CHANGING OCCASIONALLY INTO A FLOPPY DISK! ALTHOUGH HE CAN'T CONTROL THE CHANGE, HE CAN WILL HIMSELF BACK INTO HUMAN FORM. WHILE STARTING TO SEARCH FOR DR. FLOTSKY, HE HAS CHANGED, AND DISCOVERED HE CAN'T FLY AS A FLOPPY!!
I CAN'T CONCENTRATE AND CHANGE BACK! I JUST HOPE I LAND ON SOMETHING SOFT!
AHHHHH!!
CLOSE ENOUGH!
THUMP!
CHIPS
NOW I CAN CHANGE BACK.
ER... HI!
!
WONDER WHAT HER PROBLEM IS?
YAHHHHHH
OH WELL, NOW TO WORK. I NEED INFO
HE BEGINS HIS WORK BY "TALKING" TO HOODS AND INFORMANTS....
DON'T KNOW!
I DON'T KNOW
UNTIL...
OKAY! I'LL TALK!
OH, I'M SO GLAD!
HE COULDN'T TELL ME WHERE FLOTSKY IS BUT HE SAID WHERE HIS ASSISTANT WORKS AND...
TONITE SOMETHING BIG IS HAPPENING THERE!
TRASH
MEANWHILE.....
HURRY! CAP'N SYNTAX IS DEPENDING ON US!
AC/DC/BC
I'M GOING AS FAST AS I CAN TO MESS UP FLOTSKY'S COMPUTER. THIS WILL HELP MAKE THE MICRO CHIPS USELESS.
DID SOMEBODY MENTION MY NAME?
FLOTSKY!
HANDS UP, GUYS!
AT THAT MOMENT...
THERE IT IS!
I HOPE I'M NOT TOO LATE!
CRASH!!
WHAT?
CEASE AND DESIST!
YOU'RE TOO LATE!
HAVE A LOOK!
I AM HIS SERVANT
THAT CHIN...
THE PRIME MINISTER?
TO BE CONTINUED?

# A Gazeteer Of Programming Languages

*The following article appeared in the November 2, 1984 edition of the University of Waterloo's mathNEWS. The author is unknown.*

### SIMPLE

'Simple' is an acronym for Sheer Idiot's Programming Linguistic Environment. This language, developed at Hanover College for Technological Misfits, was designed to make it impossible to write code with errors in it. The statements are, therefore, confined to 'begin', 'end', and 'stop'. No matter how you arrange the statements, you can't make a syntax error.

Programs written in Simple do nothing useful. They thus achieve the results of programs written in other languages without the tedious, frustrating process of testing and debugging.

### SLOBOL

Slobol is best known for the speed, or lack of it, of its compiler. Although many compilers allow you to take a coffee break while they compile, Slobol compilers allow you to travel to Bolivia to pick the coffee. Forty–three programmers are known to have died of boredom sitting at their terminals while waiting for a Slobol program to compile.

### VALGOL

From its modest beginnings in Southern California's San Fernando Valley, Valgol is enjoying a dramatic surge of popularity across the industry.

Valgol commands include 'really', 'like', 'well', and 'y*know'. Variables are assigned with the '=like' and '=totally' operators. Other operators include the California Booleans, 'fersure' and 'noway'. Repetitions of code are handled in 'for/sure' loops. Here is a sample Valgol program:

```
like y*know (I mean) start
if pizza = like bitchen and
   b = like tubular and
   c = like grodyax
then
   for I = like 1 to oh maybe 100
      do wah - (ditty)
      barf(1) = totally gross (out)
   sure
like bag this problem
really
like totally (y*know)
```

Valgol is characterized by its unfriendly error messages. For example, when the user makes a syntax error, the interpreter displays the message:

gag me with a spoon

### LITHP

This otherwise unremarkable language is distinguished by the absence of an 's' in the character set. Programmers must substitute 'th'. Lithp is said to be useful in prothething lithtth.

### LAIDBACK

Historically, Valgol is a derivative of Laidback, which was developed at the (now defunct) Marin County Center for T'ai Chi, Mellowness, and Computer Programming, as an alternative to the intense atmosphere in nearby Silicon Valley.

The centre was ideal for programmers who liked to soak in hot tubs while they worked. Unfortunately, few programmers could survive there for long, since the centre outlawed pizza and RC Cola in favour of bean curd and Perrier.

Many mourn the demise of Laidback because of its reputation as a gentle and non–threatening language. For example, Laidback responded to syntax errors with the message:

Sorry, man, I can't deal behind that

### C–

This language was named for the grade received by its creator when he submitted it as a project in a university graduate programming class. C– is best described as a 'low–level' programming language. In general, the language requires more C– statements than machine–code instructions to execute a given task. In this respect it is very similar to COBOL.

### SARTRE

Named after the late existential philosopher, Sartre is an extremely unstructured language. Statements in Sartre have no purpose; they just are. Thus Sartre programs are left to define their own functions. Sartre programmers tend to be boring and depressed and are no fun at parties.

### DOGO

Developed at the Massachusetts Institute of Obedience Training, Dogo heralds a new era of computer–literate pets. Dogo commands include 'sit', 'stay', 'heel', and 'roll over'. An innovative feature of Dogo is 'puppy graphics', a small cocker spaniel that occasionally leaves deposits as he travels across the screen.

*And this one from Nick Sullivan . . .*

### Lingua Programatica

As a programmer who has frequently been frustrated by the lack of flexibility of conventional high–level programming languages, I am pleased to report the recent completion of a new language that promises to leave Pascal and the others stumbling in its tailwind. The new language is called LATIN (not to be confused with the natural language, Latin, with which it is, however, identical).

LATIN offers such conveniences as Roman numeral mode (for those who are tired of trying to deal with clumsy Arabic numbers), output to marble, and a sophisticated user interface that features not just icons but also omens. The package includes complete error detection and punishment. Program execution is rapid; however, programmer execution is painfully slow.

The carefully written documentation is hand–copied on papyrus scrolls by Egyptian slaves, and scans nicely. The language is provided on a sturdy double–sided discus, designed for years of trouble–free use.

Availability of LATIN is something of a problem at present, as the compiler is written not in assembler but in an intermediate–level language called GREEK (G–Code), which has yet to be implemented on any microcomputer.

*And this one by Karl Hildon . . .*

### NORTH

NORTH programs can only execute efficiently where snow falls at least 5 months of the year. This is because many NORTH programmers become sick up and fed with their environment and move on to SOUTH. Almost all NORTH programs are totally useless in the SOUTH environment.

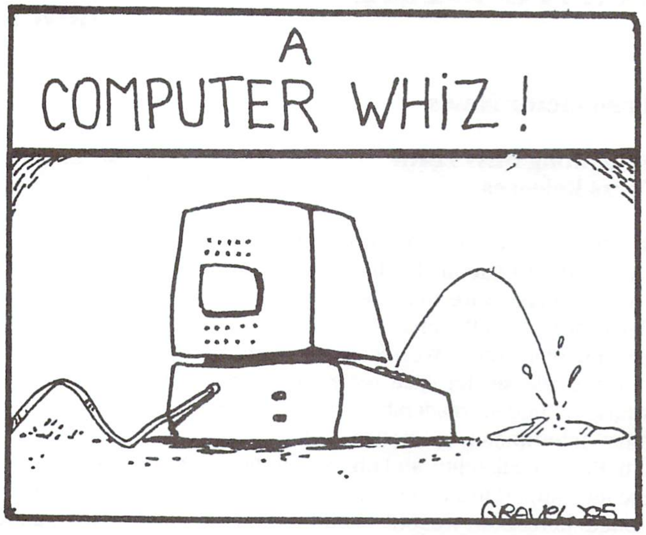
NORTH programs are immediately recognizable by the " , eh " suffix which seems to be necessary after every line. Although there are other slight differences, most NORTH programs can be translated to SOUTH by replacing the " , eh " suffix with " , uh ".

Debugging NORTH programs is no probs. The "Gimme a break" command can be inserted to stop programs from taking off with goofs, and after an error, the "Check it out" command shows the offending botches.

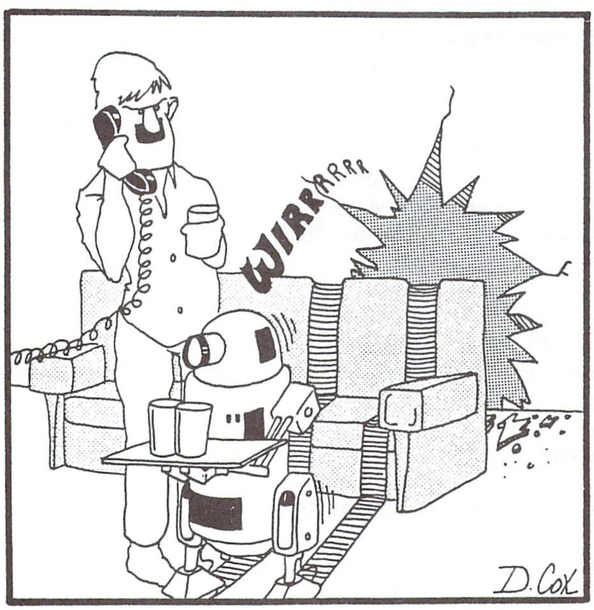
The following is a demo program that comes with the NORTH interpreter:

```
10 hosers = 1, eh
20 buzz hoser, "what's happenin man?", eh
30 far out, eh : hosers = hosers + 1, eh
40 if hosers < beer/6 then 20, eh
50 if dough = 0 then cruise, eh : goto 50, eh
60 if donuts = 0 then cruise, eh
70 if beer < 24 then cruise, eh : beer = beer + 24, eh
80 killer, eh
90 on stereo goto heavy metal, heavy metal, heavy metal
100 while beer > 0, eh
110 beer = beer - hosers, eh
120 endwhile, eh
130 if munchies then do food, eh
140 if burnt out then crash, eh : else 70, eh
```

## Compu-toons








"Efficient?. . . Oh yes, it's efficient!
Maybe a little TOO efficient."

# News BRK

**Please Note: The Transactor has a new phone number: (416) 878 8438**

## Transactor News

### Submitting NEWS BRK Press Releases

If you have a press release which 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 to the trash bin. Price, availability, and phone numbers are also important. It should also be mentioned here that we only print product releases of specific interest when related somehow to Commodore equipment.

### The Worst Kind Of Crash

Normally here at The Transactor we like to hear about new kinds of crashes. Not this time. John Mostacci, Art Director at The Transactor, had the ultimate misfortune of experiencing an auto mishap of the far-worse-than-fender-bender type variety. Photos of John's car (which now looks more like a slice of pizza with a bite taken out of it) would make great material for a fairly gruesome tale.

You'll be glad to know John is ok except for just enough damage to render him officially incapacitated. A broken forefinger to his right and a nasty gash on his left, not to mention a merciless blow to the knee and other assorted gouges, meant this months cover would require a contingency plan. I'd like to thank Carlo Mostacci for coming to the rescue. Fortunately for us, two artists were slated for the Mostacci family, and fortunately for Carlo his supervisor had two taped up hands (Fortunately for me they both have a sense of humor, right guys? I said, right guys?).

John should be back to the brush for the next cover, but until then, on behalf of The Transactor staff and readers, "Get well soon, John, we miss you".

## Events

### PCCFA – Computers In Action

It is with great pleasure that we announce the sixth annual Pacific Coast Computer Fair, Computer In Action, to be held September 14 and 15, 1985, at the Robson Square Media Centre, Vancouver, B.C.

Ours was the first personal computer fair held in western Canada and is unique as the only major Canadian fair presented by a non–profit association. Each year it draws from five to eight thousand visitors.

One of the most exciting aspects of the Fair is our speakers program. This year we will again have over two dozen speakers, including:

- Alan Boyd, Director of Software Acquisition, Microsoft
- Jim Button, author of PC–File III
- Andy Hertzfeld, principal software architect of the Apple Macintosh
- Tim Paterson, co–author of MS–DOS 1.1
- Bob Wallace, author of Microsoft Pascal and PC–Write

The talks, panels, and workshops presented will cover a wide range of topics related to personal computing. These will include:

- Artificial intelligence
- How to write for computer publications
- Local area networking
- Logo
- Purchasing computer books
- Purchasing computer software
- Telecommunication
- Unix

For more information, please contact:

Susan Brenan
Pacific Coast Computer Fair Association
P.O. Box 80866
South Burnaby, B.C.
V5H 3Y1    604 581–6877

### ISECON '85 – The Information Systems Education Conference

ISECON, sponsored by the Data Processing Management Association Education Foundation (DPMA–EF), will be held October 26th & 27th, 1985, at The Sheraton Houston Hotel in Houston, TX.

This years' theme is Dissemination of Information Systems (IS).

More than sixty presentations and panel discussions on topics of major concern to IS professionals; exhibits presented by major publishers and manufacturers of hardware, software, and audio/visual delivery systems; DPMA Special Interest Group of Education (EDSIG) Educator Award presentation; computer film and video tape festival; keynote speaker – IBM Fellow Dr. Harlan Mills, and nationally recognized luncheon speaker.

Who should attend: Computer systems education; undergraduate instructors with majors in data processing, computer science and management information systems; business professionals with interest in computer information systems; and future IS professionals. For more information, contact:

ISECON '85
Data Processing Management Association
505 Busse Highway
Park Ridge, IL
60068–3191    312 825–8124

### Western Ontario Business/ Computer Show and Seminar

December 2nd, 3rd & 4th
City Centre Complex
Commonwealth Ballroom
Holiday Inn
London, Ontario

A sales success story you can put to work for you! If you sell. . . business machines, computer hardware, computer software, office furniture, office copiers, typewriter/ word processors, filing systems, business telephone systems, office supplies and services, any products in the computer and business technology line. . . the Show & Seminar should be a vital part of your sales strategy.

Plus. . . we arrange a free seminar series on up–to–the–minute business trends – a proven drawing card that will bring qualified sales leads directly to you.

To reserve space or obtain further information, contact:

Don Young, Exhibits Manager
Brian Jones, Show Manager
Southex Exhibitions
1450 Don Mills Road
Don Mills, Ontario
M3B 2X7    416 445–6641

## Evolution of the Digital Pacific

PTC '86, the 8th Annual Forum of the Pacific Telecommunications Council, will continue the discussion of telecommunications for Pacific development. The conference will be held January 12th - 15th, 1986, Hawaiian Regent Hotel, Waikiki, Honolulu Hawaii.

Three sub-themes of PTC '85 will examine 1) Current telecommunications developments in the Pacific; 2) Future developments including computer communication convergence, artificial intelligence, ISDN; 3) Training & Education needs and programs relevant to current and future needs.

PAPERS are requested in each of the three sub-themes.

1. Current developments will cover a broad spectrum including facilities developments, business aspects, user needs an concerns, regulatory and policy questions, standards, economics. Focus may e on voice, data, video and broadcast topics.

2. Future developments will focus on probable implementations will will impact telecommunications and societies in the 1990's and beyond. Papers should focus on the technological aspects as well as on the possible impact - social, economic, education.

3. Overviews of existing telecommunication training organizations and programs including discussions of how program relate to perceived future needs of trainees and users.

Papers written jointly by persons from different countries are encouraged. Please submit a one page outline of your proposed paper to PTC '86.

DEADLINES: Outlines for proposed papers must be received by June 15, 1985. Notification of acceptance/non acceptance will be given August 1st, 1985. First full drafts will be due September 30th, 1985. Final manuscript will be due November 30th, 1985.

EXHIBITS related to the conference themes are especially invited. For PAPERS, EXHITITS or INQUIRIES, please contact:

Richard J. Barber, PTC Executive Director
Jan C. Goya, PTC Secretary
PTC '86
1110 University Avenue, #308
Honolulu, HI 96826   808 941-3789

# Books

## Four New Books from Abacus

COMPILER BOOK for the C64 & C128

The Compiler Book illustrates how a computer can transform a high-level language into machine-executable code. The reader will also learn how to design a language suited to his problems and write a corresponding compiler. It's not only for those who need to understand or write compilers, but also for those who want to know more about how their computer works. Also included as a complete assembler and disassembler, and an introduction to the 6510 machine language commands.

CAD for the C64 & C128

This book offers a detailed and an easy-to-understand introduction into the fascinating world of Computer Aided Design. Many examples and programs included as we cover topics on 3-Dimensional drawing, reflection, duplications, zoom, and filling and much more. The reader will learn how to use the full capacity of his C64 or C128 by designing, calculating, drawing and documenting object.

MORE TRICKS AND TIPS

This book is the second volume of important techniques to aid the reader in programming on the Commodore 64. Topics covered include software protection; extending BASIC commands; character, sprite and multicolor graphics; interrupts; the kernal and operating system and others as well. With these helpful tips, the reader will enhance the usefulness of the Commodore 64.

Presenting The ATARI ST

Jack Tramiel has launched the ATARI ST - his third major product for the home computer market. As with his highly successful VIC-20 and record-shattering Commodore-64, the new ATARI ST promises to break current price/performance barriers to become the computer that brings the user "power without the price."

The book Presenting the ATARI ST give you an in depth look at this much publicized computer. Lothar Englisch and Jorg Walkowiak, two computer experts and best-selling authors examine this fascinating computer. Based upon hands on experience with the ST, they examine the fantastic capabilities of the ST - from the design of the hardware to the sophisticated operating system.

As with other ABACUS books, Presenting the ATARI ST will be sure to give complete coverage of the subject.

For more information contact:

Abacus Software, Inc.
2201 Kalamazoo S.E.
P.O. Box 7211
Grand Rapids, MI
49510   616 241-5510

## How To Write Papers And Reports About Computer Technology

A new book in the ISI Press Professional Writing Series in now available to help computer professionals write effective documentation, proposals, specifications, reports, and papers. The book covers a large number of topics including: What makes a good user manual? How do you define your audience? What techniques work best for getting information through interviews? How do you write proposals that work? How can you incorporate graphics into your writing?

The author, Charles H. Sides, is a lecturer in the Massachesetts Institute of Technology's Writing Program. His feeling is that communication is a vitally important function for every computer professional and that writing is sorely neglected during most professional training. His book fills the void. Written in a lively, readable style, the book helps remove the mystique and aggravation from professional writing responsibilities; it belongs on the desk of everyone in the computer industry who needs to write.

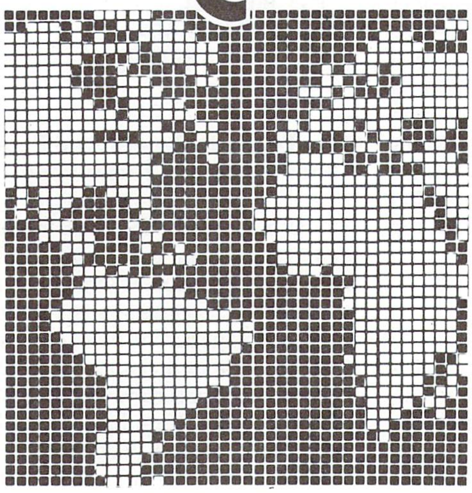
How To Write Papers And Reports About Computer Technology (162 pages) is available as a paperback (ISBN 0-89495-035-5) for $21.95. It is available at local booksellers or direct from ISI Press. Prepaid orders are shipped postpaid; billed orders are charged shipping and handling. Orders may be placed toll-free by calling 800 523-1850, ext. 1399.

Review and examination copies are available for reviewers, journalists, and educators considering the book for adoption, and may be obtained by calling 215 386-0100, ext. 1302.

Additional information may be obtained by writing to:

ISI Press
3501 Market Street
Philadelphia, PA 19104

# THE WORLD OF COMMODORE III



The 1984 Canadian World of Commodore show was the largest and best attended show in Commodore International's history. Larger than any other Commodore show in the World and this year's show will be even larger.

World of Commodore III is designed specifically to appeal to the interests and needs of present and potential Commodore owners.

Everything about your present or future Commodore computer – from hardware to software, Business to Personal to Educational – from over 90 International Exhibitors. Price of admission includes free seminars, clinics, contests and free parking.

**INTERNATIONAL CENTRE**

**DECEMBER 5 TO 8/85**

**TORONTO**

A HUNTER NICHOLS PRESENTATION
**For more information call:**
**(416) 439-4140**

# JOIN TPUG

## The largest Commodore Users Group

Benefit from:

Access to library of public domain software for C-64, VIC 20 and PET/CBM

Magazine (10 per year) with advice from
Jim Butterfield
Brad Bjomdahl
Liz Deal

TPUG yearly memberships:

| | |
|---|---|
| Regular member (attends meetings) | —$35.00 Cdn. |
| Student member (full-time, attends meetings) | —$25.00 Cdn. |
| Associate (Canada) | —$25.00 Cdn. |
| Associate (U.S.A.) | —$25.00 U.S. |
| | —$30.00 Cdn. |
| Associate (Overseas — sea mail) | —$35.00 U.S. |
| Associate (Overseas — air mail) | —$45.00 U.S. |

FOR FURTHER INFORMATION:
Send $1.00 for an information catalogue
(tell us which machine you use!)

**To: TPUG INC.**
**DEPT. A,**
**1912A AVENUE RD., SUITE 1**
**TORONTO, ONTARIO**
**CANADA M5M 4A1**

## COMAL INFO

**If you have COMAL—**
**We have INFORMATION.**

**BOOKS:**

- COMAL From A To Z, $6.95
- COMAL Workbook, $6.95
- Commodore 64 Graphics With COMAL, $14.95
- COMAL Handbook, $18.95
- Beginning COMAL, $22.95
- Structured Programming With COMAL, $26.95
- Foundations With COMAL, $19.95
- Cartridge Graphics and Sound, $9.95
- Captain COMAL Gets Organized, $19.95
- Graphics Primer, $19.95
- COMAL 2.0 Packages, $19.95
- Library of Functions and Procedures, $19.95

**OTHER:**

- COMAL TODAY subscription, 6 issues, $14.95
- COMAL 0.14, Cheatsheet Keyboard Overlay, $3.95
- COMAL Starter Kit (3 disks, 1 book), $29.95
- 19 Different COMAL Disks only $94.05
- Deluxe COMAL Cartridge Package, $128.95 (includes 2 books, 2 disks, and cartridge)

**ORDER NOW:**

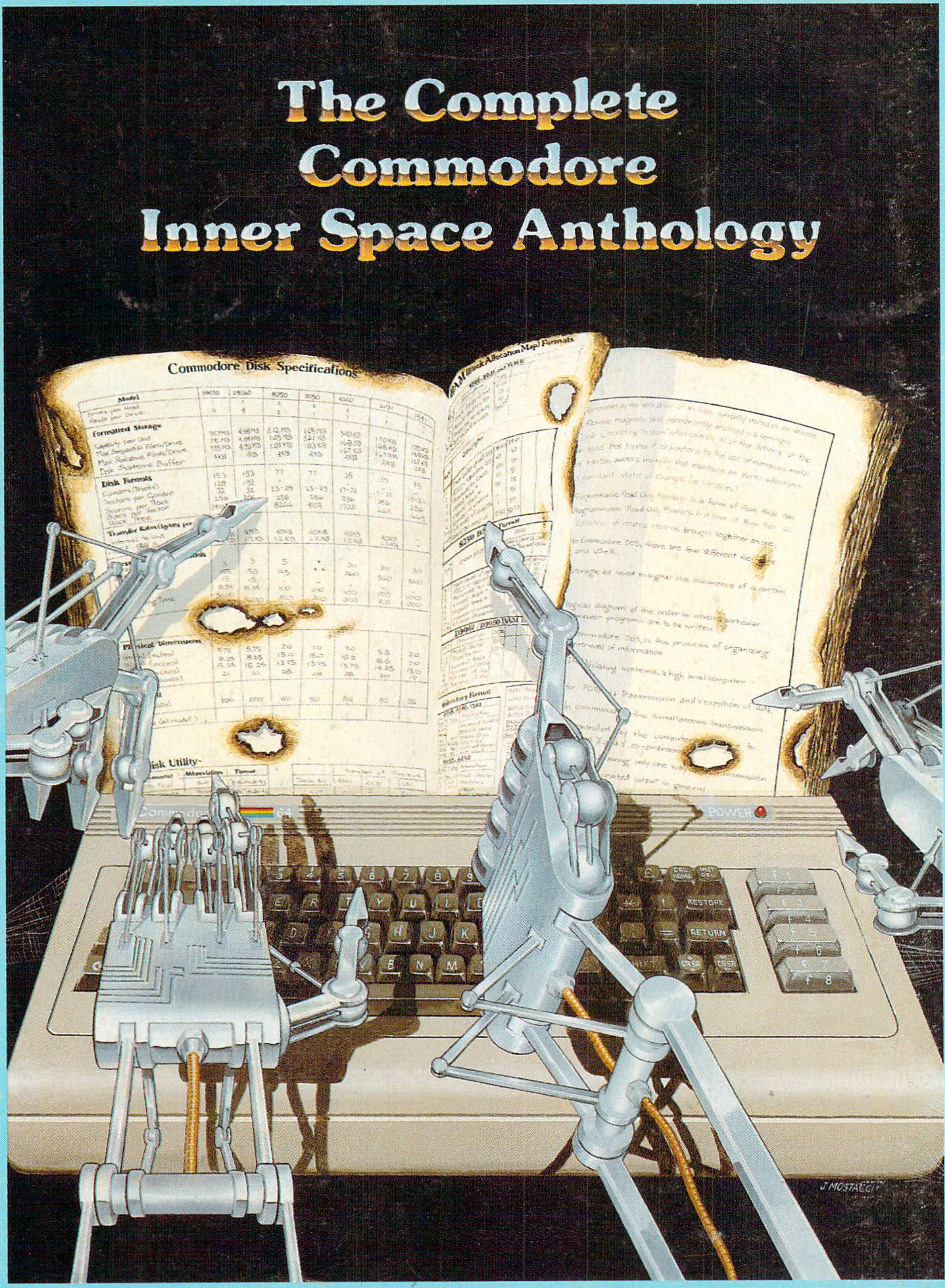
Call TOLL-FREE: 1-800-356-5324 ext 1307 VISA or MasterCard ORDERS ONLY. Questions and Information must call our Info Line: 608-222-4432. All orders prepaid only—no C.O.D. Add $2 per book shipping. Send a SASE for FREE Info Package or send check or money order in US Dollars to:

**COMAL USERS GROUP, U.S.A., LIMITED**
5501 Groveland Ter., Madison, WI 53716

TRADEMARKS: Commodore 64 of Commodore Electronics Ltd.; Captain COMAL of COMAL Users Group, U.S.A., Ltd.


The Transactor
Disk
Has Arrived!
Simply code your co-ordinates onto the postage powered order form and every program from each issue will be locked in, energized, and transported from our starbase directly to yours! Warp 9 will seem slow compared to the time you save typing, and the programs will give your machine that look and feel of a fresh set of Dilithium Crystals! Coast through the Neutral Zone with The Transactor Disk!
Only $7.95 Each!
6 Disk Subscription
Just $45.00!
J. Mostacci

# The Transactor presents, The Complete Commodore Inner Space Anthology




## Only $14.95

**Postage Paid Order Form at Center Page**