Canada $4.25
USA $3.50

# Transactor

THE MAGAZINE FOR COMMODORE PROGRAMMERS December 1988: VOLUME 9, ISSUE 2

- 256K memory expansion for your 64!
- High-performance ML graphics routines
- C-128 CP/M+ memory maps
- Cycle counting for machine language programs
- Using RS-232 printers on the 64
- Jim Butterfield on C-128 windows
- The ML Column: division routines
- Computing the distance between stars
- Re-programming the 1581
- **Reviews:** Three C128 assemblers - Merlin, LADS and Buddy; the Final Cartridge vs. Action Replay Mk. IV; BrainStorm, BrainPower and Story Writer

*Leaf Dragon* by Wayne Schmidt

# BRiWALL

Our Third Year In Business! We Know How To Service Our Customers!

## SOLID PRODUCTS & SOLID SUPPORT

P.O. Box 129 / 58 Noble Street
Kutztown, PA 19530

## 24 HOURS — TOLL FREE
## 1-800-638-5757
**OUTSIDE USA CALL 1-215-683-5433**

SATISFACTION GUARANTEED
Earn Bonus $$
No Surcharge On Charge Orders
Friendly Service

### ACCESSORIES

| | |
|---|---|
| 40/80 Column Switch Cable | $ 18 |
| Apro Extender-Cartridge Port | 17 |
| Apro Extender-User Port | 19 |
| Aprospan 4 Slot Cartridge Holder | 25 |
| Cover 1541 | 7 |
| Cover 1571 | 9 |
| Cover C128 | 9 |
| Cover C64 | 8 |
| Cover C64-C | 8 |
| Disk Storage Case-75 Disk | 11 |
| Diskettes 5½ Floppy/10 pak | 4 |
| Drive Box | 27 |
| Final Cartridge V3 | 49 |
| Hotshot Plus Interface | 69 |
| Joystick-Bat Handle | 22 |
| Joystick-Ergo Stick | 22 |
| Joystick-Super 3 Way | 25 |
| Joystick-The Boss | 16 |
| Lightpen-Model 170c | 69 |
| Lightpen-Model 184c | 49 |
| M-3 Mouse 64/128-Proportional | 39 |
| Mach 128 Cartridge | 35 |
| Mousepad | 8 |
| Paper Banner Band 45 ft. Colors | 10 |
| Paper Banner Band 45 ft. Christmas | 10 |
| Paper Banner Band 45 ft. Party | 10 |
| Paper Parchment Color | 10 |
| Power Supply-C128-Repairable | 55 |
| Power Supply-C64-Repairable | 39 |
| Quick Brown Box 64K | 99 |
| RGB to RCA Cable | 9 |
| RS232 Deluxe Interface | 39 |
| Ribbons for Printer | Call |
| Security Analyst 128 | 35 |
| Serial Box 2 for 1 | 29 |
| Serial Box 4 for 1 | 39 |
| Serial to Serial Cable-6 ft. | 9 |
| Super Chips 128 | 45 |
| Super Chips 128-D | 45 |
| Super Chips 64 | 25 |
| Super Chips 64 mode on 128 | 25 |
| Super Graphix Gold Printer Intrfc | 89 |
| Warpspeed 128 | 35 |

### BOOKS

| | |
|---|---|
| 1541 Troubleshoot & Repair Guide | 17 |
| 1571 Internals Book | 17 |
| Anatomy of C64 | 17 |
| Anatomy of the 1541 Book | 17 |
| Basic 7.0 Internals Book | 19 |
| Basic 7.0 for C128 Book | 10 |
| Beginner's Guide to BASIC 8 | 15 |
| C128 Assembly Language Prog Book | 14 |
| C128 Internals Book | 17 |
| C128 Programmer's Ref. Book | 20 |
| C128 Troubleshooting & Repair Book | 17 |
| C64 Basic Programming | 25 |
| C64 Programmer's Reference Guide | 17 |
| C64 Tips & Tricks Book | 17 |
| C64 Troubleshoot & Repair Guide | 17 |
| C64/128 Asmbly Lang. Prog Book | 14 |
| GEOS Inside & Out Book | 17 |
| GEOS Programmer's Ref. Guide | 17 |
| GEOS Tips & Tricks Book | 15 |
| K Jax Book Revealed 2 | 23 |
| K Jax Book Revealed 1 | 23 |
| Machine Language 64 Book | 13 |
| Superbase The Book 64/128 | 15 |
| Troubleshoot & Repair C64 Book | 18 |
| Twin Cities 128 Compendium 1 | 15 |

### GENERAL PRODUCTIVITY

| | |
|---|---|
| Business Form Shop/64 | 30 |
| CMS Accounting/128 | 129 |
| CMS Accounting/64 | 129 |
| CMS Inventory Module 128 | 53 |
| Chartpak 128 | 29 |
| Chartpak 64 | $ 29 |
| Evelyn Wood Dynamic Reader | 17 |
| KFS Accountant 128 | 109 |
| Leroy's Cheatsheets | Call |
| Leroy's Label Maker | 24 |
| Microlawyer/64 | 39 |
| Paperclip Publisher | 39 |
| Partner 128 | 39 |
| Partner 64 | 30 |
| Personal Portfolio Manager 128 | 42 |
| Personal Portfolio Manager 64 | 29 |
| Pocket Superpak 2 (Digital) | 67 |
| Sylvia Porter's Financial Plan 128 | 39 |
| Technical Analysis System 128 | 42 |
| Technical Analysis System 64 | 29 |
| Timeworks Account Payable 64 | 39 |
| Timeworks Accounts Receiveable 64 | 39 |
| Timeworks General Ledger 64 | 39 |
| Timeworks Inventory Managemnt 64 | 39 |
| Timeworks Payroll 64 | 39 |
| Timeworks Sales Analysis 64 | 39 |

### DATABASES

| | |
|---|---|
| Datamanager 128 | 39 |
| Datamanager 2 (64) | 17 |
| Fleet Filer 64/128 | 29 |
| Pocket Filer 2 | 29 |
| Superbase 128 | 49 |
| Superbase 64 | 42 |
| Suprbse/Suprscrpt/Book 128 PACK | 69 |
| Suprbse/Suprscrpt/Book 64 PAK | 59 |

### EDUCATIONAL

| | |
|---|---|
| Alphabet Zoo | 18 |
| Alphabuild/64 | 6 |
| Calculus by Numeric Methods | 27 |
| Counting Parade | 6 |
| Early Learning Friends/64 | 6 |
| Easy Sign | 17 |
| Facemaker/64 | 16 |
| First Men on Moon Math/64 | 19 |
| Grandma's House/64 | 6 |
| Hayden SAT Preparation | 32 |
| Jungle Book (Reading 2) | 19 |
| Kids on Keys | 18 |
| Kidsword/64 | 35 |
| Kidwriter/64 | 22 |
| Kindercomp/64 | 18 |
| Linkword: German | 19 |
| Linkword: French | 19 |
| Linkword: French 2 | 19 |
| Linkword: Italian | 19 |
| Linkword: Russian | 19 |
| Linkword: Spanish | 19 |
| Little Computer People | 12 |
| Mathbusters/64 | 6 |
| Mavis Beacon Teaches Typing/C64 | 30 |
| Peter & Wolf Music/64 | 19 |
| Peter Rabbit (Reading 1) | 19 |
| Protutor Accounting 128 | 69 |
| R.S.V.P. | $ 21 |
| Sky Travel | 35 |
| Stickybear ABC's | 22 |
| Stickybear Math 1 | 22 |
| Stickybear Math 2 | 22 |
| Stickybear Numbers | 22 |
| Stickybear Opposites | 22 |
| Stickybear Reading 1 | 22 |
| Stickybear Reading Comprehension | 22 |
| Stickybear Spellgrabber | 22 |
| Stickybear Townbuilder | 22 |
| Stickybear Typing | 22 |
| Success w/Math Add & Sub | 23 |
| Success w/Math Add/Sub Decimals | 23 |
| Success w/Math Add/Sub Fractions | 23 |
| Success w/Math Mult/Div | 23 |
| Success w/Math Mult/Div Decimals | 23 |
| Success w/Math Mult/Div Fractions | 23 |
| Toy Shop 64 | 22 |
| WhereinEurope is Carmen Sandiego | 29 |
| Where in USA is Carmen Sandiego | 29 |
| Where in World is Carmen Sandiego | 29 |
| Widham Classics-Alice/Wonderland | 12 |
| Widham Classics-Below the Root | 12 |
| Widham Classics-Swiss Famly Robs | 12 |
| Widham Classics-Treasure Island | 12 |
| Widham Classics-Wizard of OZ | 12 |

### GEOS

| | |
|---|---|
| Becker Basic for GEOS 64 | 35 |
| Desk Pak Plus-Geos | 22 |
| GEOS 64 v 2.0 | 39 |
| GEOS Programmer/64 | 45 |
| GEOS Write Workshop/64 | 39 |
| GEOS Write Workshop/128 | 45 |
| Geocalc 128 | 45 |
| Geocalc/64 | 35 |
| Geofile 128 | 45 |
| Geofile/64 | 39 |
| Geopublish 64 | 45 |
| Geos/128 | 45 |
| Geospell 64/128 | 19 |
| Fontpak Plus | 22 |
| Wordpublisher 64/128 (for GEOS) | 28 |

### CREATIVITY

| | |
|---|---|
| Advanced Art Studio/64 | 25 |
| Animation Station | 65 |
| Award Maker Plus/C64 | 29 |
| Billboard Maker/64 | 25 |
| Blazing Paddles | 25 |
| Bumpersticker Maker | 35 |
| Business Card Maker | 29 |
| Button & Badge Maker | 39 |
| Cad 3D/64 | 39 |
| Cadpak 128 | 42 |
| Cadpak/64 | 29 |
| Certificate Maker | 17 |
| Colorez 128 New Version | 12 |
| Computer Eyes | 105 |
| Create a Calendar | 22 |
| Doodle/64 | $ 29 |
| Flexidraw 5.5/64 | 29 |
| Flexifont/64 | 24 |
| Graphics Art Disk 1-10 each | 13 |
| Graphics Galleria Pak 1 | 24 |
| Graphics Galleria Pak 2 | 24 |
| Graphics Intergrator 2/64 | 24 |
| Home Designer | 45 |
| Home Designer/Circuit Symbol Lib | 10 |
| Icon Factory/64 | 25 |
| Label Wizard | 21 |
| Newsroom | 17 |
| PC Board Maker 64 | 79 |
| Perspectives II/64 | 39 |
| Photo Finish | 19 |
| Postcards | 19 |
| Printmaster-Fantasy Art Gallery | 15 |
| Printmaster Gallery I/64 | 15 |
| Printmaster Gallery II/64 | 15 |
| Printmaster Plus/64 | 25 |
| Printshop/64 | 35 |
| Printshop Companion | 27 |
| Printshop Holiday Edition | 19 |
| Screen F/X | 25 |
| Sketchpad 128 NEW | 24 |
| Slideshow Creator | 13 |
| Video Title Shop w/Gr Comp 1 | 25 |

### OUR POLICY

Our policy is to stock what we advertise and carry the best products available for your C64 and C128 computers. Over 400 of the best productivity, educational & technical software and a host of accessories in stock now! And CHECK OUT OUR C128 LINE! You will be amazed at the number of products that we carry for this fantastic machine!

### GARAGE SALE

| | |
|---|---|
| C Power 128 | 15 |
| Cover 1541 | 4 |
| Cover C64 | 4 |
| Decision Maker | 10 |
| Flexi-Aided Design | 15 |
| Forecaster-Brainpower | 10 |
| GEOS-Get Most Disk | 3 |
| Maps of Europe/64 | 12 |
| Maps of USA/64 | 12 |
| Maps of the World/64 | 12 |
| Numbers at Work | 10 |
| Project Planner | 10 |
| Tempo Type | 12 |
| Wordpro 128 | 15 |

### PERSONAL

| | |
|---|---|
| Alf | 12 |
| Bazooka Bill | 15 |
| Blackjack Academy/C64 | 27 |
| Christmas Classics/64 | 10 |
| Boston Bartender's Guide | 12 |
| Bridge 5.0 | 22 |
| Cardio Exercise & Heartlab Combo | 85 |
| Celebrity Cookbook/64 | 19 |
| Chessmaster 200/64 | 30 |
| Crossword Magic | 39 |
| Dr. Ruth's Book of Good Sex | 22 |
| Dream Machine Analyze/64 | 19 |
| Family Tree 128 | 39 |
| Family Tree 64 | 39 |
| Jeopardy 2 | 12 |
| Memory Academy 128 | 15 |
| Micro Kitchen Companion | 29 |
| Monopoly C64 | $ 25 |
| Muscle Development Package | 54 |
| Scrabble | 29 |
| Scrupples | 29 |
| Sexual Edge/64 | 19 |
| Stress Reduction-Enhanced | 299 |
| Stress Reduction-Standard | 89 |
| Strider's Classic 1-10 each | 5 |
| Tarot 128 | 15 |
| Wheel of Fortune 2 | 12 |

### SPREADSHEETS

| | |
|---|---|
| Pocket Planner 2 | 29 |
| Swiftcalc 128 w/Sideways | 39 |
| Swiftcalc 64 w/Sideways | 17 |
| Vizistar 128 | 60 |

### UTILITIES

| | |
|---|---|
| 1541/1571 Drive Alignment | 25 |
| Assembler/Monitor/64 | 29 |
| Basic 8 | 25 |
| Basic Compiler/128 | 42 |
| Basic Compiler/64 | 29 |
| Big Blue Reader 64/128 | 32 |
| Bobsterm Pro/128 | 42 |
| Bobsterm Pro/64 | 35 |
| CP/M Kit | 22 |
| Cobol/128 | 29 |
| Cobol/64 | 29 |
| Gnome Kit 64/128 | 29 |
| Gnome Speed Compiler 128 | 29 |
| Merlin 128 | 45 |
| Merlin 64 | 35 |
| Physical Exam 1541 | 29 |
| Physical Exam 1571 | 29 |
| Power Assembler 64/128 | 30 |
| Power C 64/128 | 30 |
| Programmers Toolbox/64 | 15 |
| Protolinc BBS 128 | 29 |
| Prototerm 128 | 12 |
| RamDOS 128 | 29 |
| Renegade | 29 |
| Super 81 Utilities for C128 | 29 |
| Super 81 Utilities for C64 | 29 |
| Super Aide 64 | 25 |
| Super C 128 | 45 |
| Super C 64 | 45 |
| Super Disk Librarian for C128 | 19 |
| Super Disk Utilities 128 | 25 |
| Super Pascal 128 | 45 |
| Super Pascal 64 | 45 |
| Super Snapshot 4 w/C128 Disable | 65 |
| Super Snapshot V4-New | 59 |
| Syntech BBS Const. Set/64 | 39 |
| Syntech BBS Games Module | 15 |
| SysRES Enhanced | 30 |

### WORPROCESSORS

| | |
|---|---|
| Bankstreet Writer | 35 |
| Fleet System 2 + /64 | 39 |
| Fleet System 4 | 52 |
| Fontmaster 128 w/Speller | 39 |
| Fontmaster II/64 | 35 |
| Paperclip 3 | 39 |
| Pocket Dictionary 64/128 | 10 |
| Pocket Writer 2 | 39 |
| Superscript 128 | 45 |
| Viziwrite 128 | 60 |
| Wordpro 128 w/Speller w/Filepro | 30 |
| Wordpro 64 w/Speller w/Turbo Load | 30 |
| Wordwriter 128 | 39 |
| Wordwriter 3 for C64 | 30 |
| Write Stuff 64 | 19 |
| Write Stuff 64w/Talker | 24 |
| Write Stuff C128 Version | 24 |

**1541/1571 DRIVE ALIGNMENT**

**1541/1571 Drive Alignment**
This excellent alignment program is a must have for every Commodore owner. Easy to use, it helps you to align your drive so that it runs just as if it were new! The simple instructional manual and on-screen help prompt you thru the alignment procedure and help you adjust the speed and head stop of your drives. It even includes instructions on how to load the alignment program when nothing else will load. Don't be caught short! We get more RED LABEL orders for this program, then any other program we sell. Save yourself the expense! Order now, keep it in your library and use the 1541/1571 Drive Alignment program regularly!! Works on the C64, C128, and SX64 for both the 1541 and 1571 drives.

**STILL ONLY $25.00**

**CHRISTMAS CLASSICS**

Enjoy your Christmas! Beautiful Christmas scenes are displayed while your computer plays your favorite holiday songs. Sing-along to Deck the Halls, Jingle Bells, Rudolph, Silent Night and more, as the lyrics to these Christmas classics scroll across the screen. Great for your holiday gatherings!

**JUST $10**

**HOLIDAY SPECIAL!**
Get your FREE
**Christmas Classics**
with any order over **$75.**
offer good until 12/31/88

**RAMDOS**
Lightning Fast
**RAM-DISK**

**RamDOS 128**
Yes...we know that Ram Expanders are scarce these days. But for you lucky ones that already have one, RamDOS 128 is just for you! This complete RAM based "Disk" Operating System for the 1750 RAM Expander, will turn ALL or part of your expansion memory into a lightning-fast RAM-DISK! Under RamDOS, a 50K file( 190 blocks) will load in just ½ second. Load entire disks or individual files into your expansion memory and get to work. When done save the entire memory or individual files back to your disk for permanent storage. RamDOS makes your work fast and easy!

**only $29.00**
(C128 and 1750 Ram Expander required)

# Home Designer

*When the details are important!*

**Home Designer CAD 128**
Given glowing ratings by every major Commodore magazine, this CAD system outclasses every other CAD program, because of its object-based design. With over 50 powerful commands, 5 drawing layers, superb support of library figures and lazer-quality printouts at ANY scale on your dot matrix printer or plotter, you can create drawings so accurate that a blueprint can be made from them!

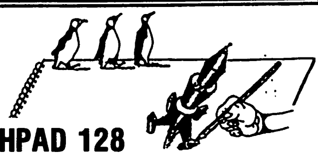
Tired of working with poor quality/inaccurate print-outs, manipulating little dots on a bit-map, giving up on detailed work because you can't zoom in close enough? Join the professionals!

**only $45.00**
mouse or joystick required
**NEW!**Engineering Library disk available now. **$10.00**

**CATALOGS**
People tell us our catalog is the BEST! Find out for yourself. Our 40 page catalog is crammed full of detailed descriptions of hundreds of C64/C128 products. Call or write for your copy today!

**BONUS DOLLARS**
EARN BONUS $$$ WHEN YOU ORDER!! For every $50 of software and accessories that you order, you earn one bonus dollar. That's an additional 2% discount!! Use your bonus dollars on future purchases.

**BRiWALL**
SOLID PRODUCTS & SOLID SUPPORT
P.O. Box 129 / 58 Noble Street
Kutztown, PA 19530
24 HOURS — TOLL FREE
1-800-638-5757



**SKETCHPAD 128**
Brand new from Free Spirit, Sketchpad 128 fully supports your C128 and takes advantage of its crisp 80 column graphics capabilities. It is packed with all the features of a professional drawing package such as drawing SMOOTH freehand lines, 3D Solids, creating Slideshows, Cut & Paste, Clip, Flip, Enlarge, Shaded Fill Patterns, a variety of Fonts, Air Brush and more! It supports Printshop graphics and is completely compatible with all BASIC 8 files.
Sketchpad 128 unleashes the graphics power of your C128! It supports your 1351 Mouse, 64K Video Chip, 1581 drive and 80 column display. **What more could any real C128 user ask for?**

**ONLY $24.00**

SUPER 81 UTILITIES

**Super 81 Utilities**
This full-featured disk utilities system, for your 1581 drive, is available in both C64 and C128 versions. Among the many features included in this fine package are:

- Copy whole unprotected 1541/1571 disks to 1581 partitions
- Copy unprotected 1541/1571 files to 1581 disks
- Copy unprotected 1581 files to 1571 disks
- Backup 1581 files or disks with 1 or 2 1581's
- 1581 Disk Editor, Drive Monitor and RAM Writer
- Includes numerous DOS commands such as Rename a file/disk,
- Scratch/Unscratch files, Lock/Unlock files, Create Auto-Boot and more

Tired of not being able to use your 1581? Super 81 will get that great little drive out of the closet and into use.

**only $29.00**
(please specify C64 version or C128 version)

**Superbase**
Precision Software
**SUPER PAK SPECIAL**

**Now the best database on the market has been combined with Superscript and Superbase The Book into one super package with one super low price.**

Though easy for the novice to use, the ability to write BASIC-like programs to set up your own custom applications, turns this system into a true database of incredible power! Combine it with SUPERSCRIPT, a sophisticated text management system, that includes a spellchecker, calculator and mailmerge facility and you have an unbeatable team! And as an added bonus, you will also get SUPERBASE THE BOOK, FREE!

**Get all three in one Super Pak!**
**C64 Version $59.00**
**C128 Version $69.00**

**GREAT ACCESSORIES FOR YOUR C64/C128**

**The Drive Box** - With some simple souldering you can make any Commodore drive switchable between devices 8, 9, 10 or 11 and also write the unnotched side of your disk - **$27.00**

**2-For-1 or 4-For 1 Serial Boxes** - Connect 2 or 4 peripherals to the box and share with another device. Multiple computers sharing one drive. Multiple printers hooked to one computer, etc. 2-For 1 Box is **$29.00.** 4-For-1 Box is **$39.00.**

**Quick Brown Box** - A great cartridge into which you can write programs, utilities or data files. What you store will remain intact, even with the cartridge unplugged or the computer turned off! 32K Box is **$79.00;** 64K Box is **$99.00.**

**M3 Mouse** - If you are looking for a reliable, well-built fully proportional mouse, stop looking! Built by Contriver, the M3 Mouse is the best we've seen. **$39.00.**

**ORDERING INSTRUCTIONS**
For your convenience, you can place a phone order 24 HOURS/7 DAYS A WEEK on our TOLL-FREE LINE 1-800-638-5757. Price, Availability and Compatability Checks are also Welcome on our order line, Monday thru Friday, 9AM-4PM EST. AFTER HOURS, Orders Only Please!! When placing an order, please specify your COMPUTER MODEL, HOME & DAYTIME PHONE NUMBER, SHIPPING ADDRESS, METHOD OF PAYMENT and ITEMS ORDERED. To help us serve you better, please have all your information, including your CHARGE CARD number, ready before you call us.

ACCEPTED PAYMENT METHODS: We gladly accept payment by: PREPAID BY PERSONAL CHECK (will not hold for clearing) or MONEY ORDER: COD (continental USA only AND $4.00 ADDITIONAL); MASTERCARD; VISA; SCHOOL PURCHASE ORDERS. All payment must be in USA DOLLARS. THERE IS NO SURCHARGE FOR CHARGE CARDS and your card is NOT charged until we ship.

**TECHNICAL SUPPORT**
We do our very best to help you with your product selections, before you order and after you receive your product. General questions, price, and compatibility with your computer, etc. will be handled by our order staff at the time you place your order. BUT if you have specific, detailed questions about a product, printer compatibility questions etc., you will get the most help from our TECHNICAL SUPPORT LINE at 215-683-5699. Call Monday thru Friday, 9AM-4PM EST, and our trained tech staff will be happy to help you.

**SHIPPING POLICY**
ALL ORDERS received before 3PM EST will normally be shipped same or next business day. Out-of-Stock items will be shipped by the same shipping method as original order, normally within 3 or 4 business days. All UPS shipments are sent SIGNATURE REQUIRED/NO DRIVER RELEASE. ADD the following shipping charges to your TOTAL software order.

UPS: GROUND-$4.00 (cont USA ONLY); AIR/RUSH-$7.00 (includes Puerto Rico/Hawaii); OVERNIGHT-$12.00 PLUS $2.50 per item (must be received by 12 NOON).

PRIORITY MAIL: USA-$4.00 (includes APO/FPO); CANADA/MEXICO-$6.00; OTHER FOREIGN-$4.00 handling PLUS Actual Shipping (minimum $12.00)

**RETURN POLICY**
We have a liberal return policy to better service your needs. Software piracy is a problem, but as long as our policy is not abused, we will continue to honor it. If within 15 days from the time you receive an item, you are not satisfied with it for any reason, you may return it to us for either a refund, exchange or open credit. REFUNDS ARE SUBJECT TO A 10% RESTOCKING FEE PER ITEM RETURNED ($5.00 MINIMUM PER ITEM). A DEFECTIVE ITEM will be replaced with the same item (NO CHARGE... of course). EXCHANGES/OPEN CREDITS will gladly be issued for the FULL PURCHASE PRICE OF THE ITEM.

Transactor
The Magazine for Commodore Programmers

Volume 9, Issue 2

# Departments and Columns



# Reviews



**About the cover:** The artwork is unmistakably that of Wayne Schmidt, long known in the Commodore graphics arena for his intricately detailed work. Wayne lives in New York, where he does commercial graphics and other art using his Commodore 64. His work has appeared in several magazines. Wayne uses a variety of software in producing his images, including some he's written himself. The cover picture is in multi-colour, not hi-res mode.


Transactors's phone number is:
(416) 764-5273
Line open Mondays, Wednesdays and Fridays ONLY

FAX: (416) 764-9262

**TOLL-FREE ORDER LINE**
**1-800-248-2719 Extension 911**
(for orders only; have your VISA or Mastercard number ready; available in the U.S. only)

*Transactor* is published bimonthly by Croftward Publishing Inc., 85-10 West Wilmot Street, Richmond Hill, Ontario, L4B 1K7. ISSN# 0838-0163. Canadian Second Class Mail Registration No. 7690, Gateway-Mississauga, Ont. US Second Class mail permit pending at Buffalo, NY. USPS Postmasters: send address changes to: Transactor, PO Box 338, Station C, Buffalo, NY, 14209.

Transactor Publishing Inc. is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names are registered trademarks of Commodore Inc.

Subscriptions:

Canada $19 Cdn.
USA $15 US
All others $21 US
Air Mail (Overseas only) $40 US

**Send all subscriptions to:** Transactor Publishing Inc., Subscriptions Department, 85-10 West Wilmot Street, Richmond Hill, Ontario, Canada, L4B 1K7, (416) 764-5273. For best results, use the postage paid card at the centre of the magazine.

**Quantity Orders:** In **Canada:** Ingram Software Ltd., 141 Adesso Drive, Concord, Ontario, L4K 2W7, (416) 738-1700. In the **USA:** IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California, 92121, (619) 481-5928; ask for Dave Buescher. Quantity orders/enquiries are also welcome from computer/software distributors in the **UK, Europe and Scandinavia.** Please contact: T.G. Hamilton (W/S) Ltd., Tel: 021-742-5359; Fax: 021-742-2190; or contact Transactor (UK) direct at Unit 2, Langdale Grove, Bingham, Notts. NG13 8SR, telephone 0949-39380. In **Australia**, contact Transactor (Australia) Pty. Limited, 35 Calder Cr., Holder, ACT 2611, Australia. Phone 61 62 883584.

Editorial contributions are welcome. Only original, previously unpublished material will be considered. Program listings and articles, including BITS submissions, of more than a few lines, should be provided on disk. Preferred format is 1541-format with ASCII text files. Manuscripts should be typewritten, double-spaced, with special characters or formats clearly marked. Photos should be glossy black and white prints. Illustrations should be on white paper with black ink only. Hi-res graphics files on disk are preferred to hardcopy illustrations when possible.





**Production**
In-house with Amiga 2000 and Professional Page

Final output by Vellum Print & Graphic Services, Inc., Toronto

**Printing**
Printed in Canada by Bowne of Canada Inc.

# Using "VERIFIZER"

*Transactor's foolproof program entry method*

VERIFIZER should be run before typing in any long program from the pages of *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; you can then check this code against the corresponding one in the printed program listing.

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

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

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

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

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

VERIFIZER will catch transposition errors like POKE 52381,0 instead of POKE 53281,0. However, VERIFIZER uses a "weighted checksum technique" that can be fooled if you try hard enough: transposing two sets of four characters will produce the same report code, but this will rarely happen. (VERIFIZER could have been designed to be more complex, but the report codes would need to be longer, and using it would be more trouble than checking the program manually). VERIFIZER ignores spaces so you may add or omit spaces from the listed program at will (providing you don't split up keywords!) Standard keyword abbreviations (like nE instead of next) will not affect the VERIFIZER report code.

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

**PET/CBM VERIFIZER (BASIC 2.0 or 4.0)**

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

**VIC/C64 VERIFIZER**

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

***NEW* C128 VERIFIZER (40 or 80 column mode)**

```
KL  100 rem save"0:c128 vfz.ldr",8
OI  110 rem c-128 verifizer
MO  120 rem bugs fixed:  1) works in 80 column mode.
DG  130 rem              2) sys 3072,0 now works.
KK  140 rem
GH  150 rem by joel m. rubin
HG  160 rem * data loader for "verifizer c128"
IF  170 rem * commodore c128 version
DG  180 rem * works in 40 or 80 column mode!!!
EB  190 ch=0
GC  200 for j=3072 to 3220: read x: poke j,x: ch=ch+x: next
NK  210 if ch<>18602 then print "checksum error": stop
BL  220 print "sys 3072,1 to enable
DP  230 print "sys 3072,0 to disable
AP  240 end
BA  250 data 170, 208,  11, 165, 253, 141,   2,   3
MM  260 data 165, 254, 141,   3,   3,  96, 173,   3
AA  270 data   3, 201,  12, 240,  17, 133, 254, 173
FM  280 data   2,   3, 133, 253, 169,  39, 141,   2
IF  290 data   3, 169,  12, 141,   3,   3,  96, 169
FA  300 data   0, 141,   0, 255, 165,  22, 133, 250
LC  310 data 162,   0, 160,   0, 189,   0,   2, 201
AJ  320 data  48, 144,   7, 201,  58, 176,   3, 232
EC  330 data 208, 242, 189,   0,   2, 240,  22, 201
PI  340 data  32, 240,  15, 133, 252, 200, 152,  41
FF  350 data   3, 133, 251,  32, 141,  12, 198, 251
DE  360 data  16, 249, 232, 208, 229,  56,  32, 240
CB  370 data 255, 169,  19,  32, 210, 255, 169,  18
OK  380 data  32, 210, 255, 165, 250,  41,  15,  24
ON  390 data 105, 193,  32, 210, 255, 165, 250,  74
OI  400 data  74,  74,  74,  24, 105, 193,  32, 210
OD  410 data 255, 169, 146,  32, 210, 255,  24,  32
PA  420 data 240, 255, 108, 253,   0, 165, 252,  24
BO  430 data 101, 250, 133, 250,  96
```

# The Standard Transactor Program Generator

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

Once the program is typed in (check the Verifizer codes as usual when entering it), save it on a disk for future use. Whenever you type in a program generator, the listing will refer to the standard generator. Load the standard generator *first*, then type the lines from the listing as shown. The resulting program will include the generator code and be ready to run.

When you run the new generator, it will create a program on disk (the one described in the related article). The generator program is just an easy way for you to put a machine language program on disk, using the standard BASIC editor at your disposal. After the file has been created, the generator is no longer needed. The standard generator, however, should be kept handy for future program generators.

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

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

# Start Address

## *Light and the End of the Tunnel*

We have good news and good news. And good news. It's great to be able to say that.

First of all, *Transactor*'s "dark night of the soul" has ended. Our forte has always been creating magazines of unsurpassed technical information for users of Commodore computers. We intend to continue in that tradition.

Now, however, the matters of printing, distribution and advertising are being carried out by Croftward Publishing Inc., which is owned by a large U.K.-based publishing company with resources in excess of those that *Transactor* could provide. (Croftward publishes *Commodore Computing International* and *Amiga User International* magazines in the U.K., among others.)

This happy arrangement means that we need only concern ourselves with editorial content on our side of things and Croftward will ensure that those materials are printed and distributed to reach you on a timely basis.

Our other good news concerns magazine content. We have two new columnists writing for *Transactor.* "The Edge Connection" is where you'll find the writings of Joel Rubin, who authored our C128 80-column Verifizer. Joel is a former CompuServe sysop and has wide experience in many corners of the computing universe. Joel examines Commodore computing from a unique viewpoint and makes for very interesting reading, as you'll see. In this issue, Joel's first column compares three C128 assemblers and discusses a number of issues pertaining to them.

Finally, by popular demand, the feature you've all been so eager to see: The ML Column. Yes, it's here - and written by Todd Heimarck no less. Todd spent some years at COMPUTE! and co-authored *Machine Language Routines for the Commodore 64/128* (which was so favourably reviewed by Mike Garamszeghy in Volume 8, Issue 4). Recognize that this is not a tutorial for beginners. This column is directed at the intermediate to advanced programmer. The first column takes a look at various algorithms for doing binary division and benchmarks them. Dig in.

While we're on the subject of advanced programming, two of the articles in this issue deliver the tools and techniques for doing fast hi-res graphics. In fact, both articles are looking for the *fastest* way to plot a pixel on a hi-res screen (though both address other issues as well). The authors seem to have come pretty close to the theoretical maximum speed for this often-vital task, but if you can shave even more cycles off their routines, we'd love to hear from you and so would they. If you're "in search of the blazing bit", have a look at *Cycle Counting* by David Sanner and *Fast Graphics Primitives* by Robert Heuhn. And make sure your monitor is well ventilated.

More good news for GEOS users: This issue we're "pushing the limits" with the C256. Paul Bosacki shows you in words and pictures how to expand your C64 to 256K *internally.* The software patch to make GEOS recognize the internal RAM drive is printed in geoProgrammer format. From the feedback I've received so far, it seems that there are a number of GEOS programmers out there who *want* the listings in geoProgrammer format. If you're new to programming the GEOS environment and are using one of the other assemblers, don't fret. Next issue we'll print a cross-reference table of GEOS labels that will make it easier for people on both sides of the fence to talk to each other.

That's a lot of good news for one issue. We're very happy to have Todd and Joel writing for us and we're sure you'll be pleased as well. We're also very happy with our new arrangement with Croftward and look forward to returning to active duty on the forefront of Commodore computing.

**Malcolm D. O'Brien**

# b i t s

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

**U.S. to Canada Mail Order**
**Steve Campbell, Mississauga, Ontario**

If you order from the States and the order is shipped by United Parcel Service (UPS), there will most likely be a $12.50 fee charged to you, as well as any customs duties.

I was never advised of this unfortunate situation until I ordered some CP/M software for my C128. I got a card in the mail from UPS at the border saying to call them (toll-free) because there was a hold-up at customs.

When I called, I was told that they always check with the receiver of the parcel before it goes through customs if the sender did not pay for the duties in advance; my dealer hadn't paid. I went ahead with the order because it was software I needed, and paid the $12.50 plus $2.41 duty (on four disks and six books).

Moral: Ask for the parcel to be sent by United States Postal Service. Unless you need it immediately, you're better off using the USPS. All things considered, UPS probably takes just as long anyway.

## 64 Bits

**Now You See Me, Now You Don't**
**Sudharshan Sathiyamoorthy, Scarborough, Ontario**

Looking for a new effect to add to your programs? Type in the following, save it, and then run it.

```
ON  100 rem fader - by sudharshan sathiyamoorthy
PB  110 for i=49152 to 49193: read a: poke i,a: next
AE  120 data 165,  78, 133,  80, 165,  79, 133
OC  130 data  81, 166,  83, 164,  82, 202, 165
OO  140 data  87, 136, 145,  80, 192,   0, 208
OC  150 data 249,  24, 169,  40, 101,  80, 133
LM  160 data  80, 169,   0, 101,  81, 133,  81
PA  170 data 164,  82, 224,   0, 208, 227,  96
```

Now type in the following example program, save it, and then run it.

```
HA  100 rem fader example
EN  110 rem    by sudharshan sathiyamoorthy
GJ  120 rem
JB  130 rem make sure fader is installed
KK  140 rem
IH  150 poke 82,20:poke 83,10:poke 53280,0:poke 53281,0
MF  160 for i=1 to 10: read c(i): next i
FG  170 data 0,11,12,15,1,1,15,12,11,0
PI  180 poke 78,0: poke 79,216
OH  190 for i=1 to 10: poke 87,c(i): sys 49152
        : for j=1 to 40: next j,i
LG  200 poke 78,164: poke 79,217
NP  210 for i=1 to 10: poke 87,c(i): sys 49152
        : for j=1 to 40: next j,i: goto 180
```

Make sure the screen is fairly full to see the effect properly. Pretty neat, isn't it? The fading is achieved by cycling the colour of a section of the screen through different shades of white, black and grey.

To use this effect in your program, append the first listing to your BASIC code. The routine is relocatable - at present it resides at address 49152. The routine also uses addresses 78 to 83 and address 87 for storage purposes.

The machine language program allows you to change the colour of a rectangular area on the screen. The top left corner of the rectangle is stored in address 78 and 79 (in low, high format) - the value used here is a colour memory address starting at 55296. The number of character rows and columns in the box is stored in addresses 83 and 82, respectively. Finally, the colour is put in location 87.

The machine language routine simply changes the colour of a section of the screen; the actual colours that you choose determines the fading effect. The example BASIC program above fades from white to black and from black to white. To fade from yellow to black and from black to yellow, modify the second program as follows:

```
160 dim c(12): for i=1 to 12: read c(i): next i
170 data 0,9,2,8,10,7,7,10,8,2,9,0
```

in lines 190 and 210, change 'for i=1 to 10' to 'for i=1 to 12'.

**Growing Print**
**Jeremy Hubble, Belton, Texas**

This little subroutine does a lot for its size: it not only centres text, but it causes it to "grow" out from the centre of the screen, and can also change the text colour. To use it, simply set A$ to the string you want to print (it should be less that 39 characters), set C to the colour you want to print in (0 through 15), and GOSUB 2000. You can change the speed by altering the value for D1 in line 2000. Lines 10 and 20 are just to demonstrate the subroutine, and can be deleted when putting the subroutine your own programs.

```
KH  10 poke 53281,0:a$="growing print": c=1:gosub 2000
CC  20 print:a$="by jeremy hubble":c=2:gosub 2000:end
DJ  2000 d1=10: poke 646,c: a=len(a$)
         : if a/2<>int(a/2)then a$=a$+" ": goto 2000
DA  2010 for b=1 to a/2
         :print tab(21-b)left$(a$,b)right$(a$,b)"{up}"
JM  2020 for d=1 to d1: next: next: print: return
```

**The RND Function**
**Evan Williams, Williams Lake, British Columbia**

I have seen, heard, and read a lot of misconceptions about how the random number generator in Commodore BASIC works or doesn't work. I have not yet seen an accurate description of what really happens. First, without getting into what randomness is, I will say that the Commodore BASIC RND function is one of the better pseudo-random number generators available. It generates very long initial sequences of numbers with very good apparent randomness. No detectable patterns appear at binary intervals up to 2^16. No significant bias exists such as for even over odd numbers.

The biggest misconception I know of regarding RND is that one can "improve" the randomness by using some number other than one as the argument (i.e. x=rnd(ti)). This has absolutely no effect as when the argument is positive it is a dummy and is not used by the RND function. The next misconception is that something is wrong with RND when negative numbers are used. Not so! Negative numbers are used to seed the RND function and initiate a new sequence. When any negative number is used in the RND function (i.e. x=rnd(-27)), this starts the random number generator at a new point. For every negative number a different sequence results. For really random output, use x=rnd(-ti) at the start of a program.

Furthermore, and even better, if the same negative number is used, the same sequence results. This allows you to control the output of the RND function, which can be useful in games (for example, to give each player the same random maze to negotiate).

Speaking of games, I should mention that a lot of software for the C64 uses the noise generator in the SID chip to create pseudo-random numbers. This is a hardware version that works using a shift register with gated feedback of the binary output. I haven't examined the characteristics of this part of the chip so I can't comment on it except to say that the more the white noise output sounds like white noise (a hissing sound), the better it probably is. I have heard some artifacts in the white noise output which sound sort of metallic and clinky so it probably is suspect. As a last note, when zero is used as an argument for RND the output *seems* okay but has gaps in the numbers it hits. These gaps are regular and periodic so zero shouldn't be used.

**Quick Block Display**
**Brian Spencer, Barrie, Ontario**

Have you ever wanted to quickly display a block from a disk onto the screen in its raw form? Run the following program and you'll see just how simple and short a task that can be.

```
JC  10 for i=828 to 844:read d:poke i,d:next i
NP  20 data 162,2,32,198,255,162,0,32,207
HC  30 data 255,157,0,4,232,208,247,96
GG  40 input"{clr}{8 down}track, sector";t,s
CF  50 open 15,8,15: open2,8,2,"#"
CL  60 print#15,"u1 2 0"str$(t)str$(s)
IE  70 sys 828:poke 780,2: sys 65475: sys 65511
```

Program explanation:

| | |
|---|---|
| Line 10 | Load in the ML GETBLOck routine |
| Lines 20,30 | ML data for GETBLOck routine. Notice the 0 and 4 that denotes the top of screen memory ($0400, or 1024 decimal) |
| Line 40 | Prompt user for track and sector |
| Line 50 | Set up channels (device 8 default) |
| Line 60 | Jump to track and sector entered by user |
| Line 70 | Call GETBLOck routine, which puts 256 bytes from logical file #2 into screen memory; use Kernal to close logical file #2 and close all files |

The files are opened and closed from BASIC so that you can modify the program without altering the machine code.

**SX-64 ROM Bug**
**Kevin Hopkins, Monticello, Illinois**

There is a bug in the Kernal ROM for the SX-64. If you use SHIFT/RUN-STOP to load and run your Quantum Link terminal software or any other programs using RS-232, this bug can cause a crash very quickly.

A comparison of the SX Kernal and the version 3 ROM of the C64 (from which it descended) reveals that there are 49 bytes different in the SX Kernal. The trouble lies with the changes in the LOAD/RUN text for the SHIFT/RUN-STOP key combination. Upon call, this text is written into the keyboard buffer. Now, most programmers understand that this buffer is only 10 bytes long. No problem with the Version 3 text which is only 9 bytes long, but the SX text is 15 bytes long, which means that five bytes are going to go where they don't belong. Where they don't belong in this case is the operating system's pointers to start of memory, end of memory, and the flag for the Kernal variable for IEEE time-out. This is exceedingly nasty.

I believe the cause of the problem is the overwriting of the end-of-memory pointer at $0283/4. *Mapping the Commodore 64* states that the start-of-memory pointer is never used after power-up (the value is copied to zero page and that copy is used thereafter), and the time-out byte is for an IEEE interface that Commodore was to release. But the end-of-memory pointer is used in the allocation of the RS-232 input and output buffers, which is done whenever an RS-232 file is opened. Writing the "UN" of the "LOAD...RUN" text into that pointer by pressing SHIFT/RUN-STOP would have you allocating buffers at $4E55 on down. One hesitates to speculate on how many things can go awry because of this re-coding error. Depending on the size of your program, this could spell instant software death.

The SX bug is cause for concern today because, although that machine is no longer made, the SX Kernel is a natural foundation for anyone creating a customized operating system for the C64. The SX code has already isolated the tape routines, which a programmer can freely replace with whatever he likes before buring a replacement ROM.

Solutions: If you are buring a new ROM, the bug can be crushed by abbreviating the command string by changing "load"*",8<cr>run<cr>" to "lO":*",8<cr>rU<cr>", which reduces the string by three of the five bytes, or by removing the ",8" and changing the default device number elsewhere in the Kernal.

## C128 Bits

**Who Needs Fast-Loaders?**
**John Fehr, Gretna, Manitoba**

The other day, I was fooling around with the 128 and 64 modes when I noticed that the memory from bank 0 in 128 mode remains intact after going into 64 mode. Experimenting, I found that it was possible to load a machine language program in 128 mode, go into 64 mode, and execute the program. Why go through the hassle? With a 1571 disk drive, loading in 128 mode is faster than loading in 64 mode with a fast-load cartridge!

To try this, boot the computer in 128 mode and load a BASIC program like this:

```
bload"filename",d0,u8
```

Reset the computer, hold down the Commodore key (to enter 64 mode), and the program will still be in memory. To "unnew" the program so that you can list and run it, type:

```
poke 2050,10: sys 42291: poke 45,peek(34)
: poke 46,peek(35): clr
```

**Defining Keys in CP/M**
**Douglas Taylor, Columbus, Ohio**

The KEYFIG command that comes on the C128 CP/M boot disk allows you to define up to 32 different function keys on the C128, each with its own string. A 'string' in this case implies two or more characters, such as 'dir [RETURN]' (assigned to F3) or 'help ' (assigned to F27). Here is a shortcut that lets you redefine keys in CP/M mode without using KEYFIG, even from within many application programs!

Suppose you want to redefine the F1 key, which defaults to the string "F1". To do this, hold down the CTRL and right-SHIFT keys and press the grey cursor-right key. A small white box will appear at the bottom of the screen. Next, press the function key you want to redefine, in this case F1. The white box will now read >F1< in reverse video with the cursor over the F. Now type in something like, "This is a function key!". If you make a mistake you can cursor back with CTRL/right-shift/cursor-left, and forward with CTRL/right-shift/cursor-right. You can also insert with CTRL/right-shift/+ and delete with CTRL/right-shift/- (the + and - keys on the main keyboard, not on the keypad). When you have completed typing the definition, press CTRL/right-shift/RETURN. Commodore chose the CTRL/right-shift combination so that you can define any key as a function key and include cursor movements in your strings, as well as inserts, deletes, and carriage returns. Now press F1. You will see the words "This is a sunction key!" printed on the screen. You can even do this from inside your favourite word processor for instant key macros.

That's all there is to it. Remember, this method only works for redefining keys which are assigned strings of two characters or more, so you couldn't use it to redefine, say, the "a" key. But once you have used KEYFIG to set up your keyboard you can have up to 32 easily redefinable function keys. I have not yet found a program in which I could not use this shortcut method, but with IMP.COM in terminal mode, the keys do not play back correctly.

By the way, all this information can be found in the CP/M section of the C128 Programmers Reference Guide.

**Window Wiper**
**Chris George, Islington, Ontario**

If you are tired of using chr$(147) to clear your screen, you can easily liven up your programs with a more creative "screen wipe" by using the "clear" option of the BASIC 7.0 window command. Try the following one-liner:

```
10 rem wipe out c128 (40/80)
20 w=rwindow(2)/2:r=3.3/(1-(w=20)):for i=0 to 12
   :window w-r*i,12-i,w+r*i,12+i,1:next
```

This program wipes the screen from the inside to the outside in either 40 or 80 column mode. Experimenting with this technique can produce some interesting results.

# The ML Column

## *What do you do, subtract a lot?*

**by Todd Heimarck**


Welcome to the ML Column. I'm sure the *Transactor* editors will correct me if I'm wrong, but I believe an unwritten rule states that "Brand new columnists are allowed to introduce themselves and their column." Another unwritten rule is this: "You can always quote an unwritten rule. How will the readers know if it's correct or not?" Another unwritten rule is this: "You can never quote an unwritten rule. As soon as you quote it, you've written it down, which negates its status as unwritten. Therefore, you're a liar."

Sounds like an old Star Trek episode where Kirk tells a computer that it made a mistake. By not catching the mistake, it made another mistake. By not catching that mistake, it made another. And so on. Computers are gullible. If you type "EVERYTHING I SAY IS A LIE" it will confuse a 64 (or even a 128). The result is ?SYNTAX ERROR, which is ridiculous when you consider that the syntax of the sentence you typed is perfect, whether or not you include the quotation marks.

But let's return to the introduction. I bought my first computer, a VIC-20 with an 8K expander and a Datassette, back in 1981 or 1982. After a year or two, I moved up to a 64. Later I bought a 1541 disk drive. When the 128 first appeared, I bought one, along with a 1571 drive and a 512K memory expander. I also own an Atari 1040ST, but we won't say anything more about that in these pages.

I've programmed in 6502 assembly language (also called "machine language") since 1983, the same year I started working at COMPUTE! Publications. There I wrote articles, wrote programs, and edited articles. If you read the *Gazette*, you might have seen the *Horizons* column. Charles Brannon (author of *SpeedScript* and other goodies) founded the *Horizons* column. I wrote it for a couple of years. It's now in the capable hands of Rhett Anderson. At various times, I also worked on *Bug Swatter*, *Hints & Tips*, and *Gazette Feedback*.

I recently moved from North Carolina to Washington state after accepting a job at Microsoft Corporation, where I write manuals describing their programming languages.

This column will focus on 6502 assembly language algorithms for the 64 and 128, with an emphasis on intermediate to advanced programming. Although I have several ideas for columns (searching, sorting, crunching, file I/O, sprites, interrupts, graphics), I'd like to hear from *Transactor* readers. If you want to suggest a topic or respond to a column, write to me c/o *Transactor* at the address in the front of the magazine. Better yet, send email on CompuServe (ID 76703,3051). You could also drop into my Thursday night conference in the CompuServe CBMART area. It happens every Thursday night from 10-12 PM Eastern time. For those of us on the left coast, that's 7-9 Pacific time. If you're in Iowa, Hawaii, or Newfoundland, you'll have to calculate the appropriate time for yourself. Log onto CompuServe, GO CBMART at a ! prompt, join the forum if you aren't a member, and type CO to enter the conferencing area. Incidentally, you can find the example programs from this column on the Transactor Disk or in the *Transactor*'s own CBMPRG forum on CompuServe.

### How to divide

Enough of the introductory stuff. Let's talk about machine language.

The other day I mentioned (in passing) to an IBM assembly language programmer that the 6502 didn't have a divide instruction and that you had to do it in software. He stopped dead in his tracks, thought deep thoughts for a minute or two, and said, "No divide instruction? What, do you just subtract a lot?"

Well, yes and no.

A little later, someone left a message in the CBMPRG area of CompuServe asking how to divide in assembly language.

How do you divide? There are four (or five) answers to the question. But first a few remarks.

### Testing the alternatives

Given four (or five) answers, you might want to test them out. Which one is fastest or most compact or most elegant? Let's invent a benchmark situation. We want to divide one number by another. Call them N1 and N2. We don't want to destroy the values in memory, so the first thing to do is copy N1 to NUM and N2 to DEN (I picked those names because in a fraction the number on top is the NUMerator and the number below is the DENominator). The second thing to do is test DEN for a value of zero (because you can't divide by zero). If something (such as division by zero) causes an error, the routine should squawk. The carry flag isn't affected by the RTS instruction, so we'll arbitrarily set the carry if an error occurs and clear the carry to indicate success.

We'll also arbitrarily decide that NUM and DEN are 16-bit values and that the result will be stored in another 16-bit variable called RES. The remainder will end up in REM. Two more obvious choices for label names. (If your variables have other sizes, you can easily modify the example programs to whatever size you need. The variable RES should be the same size as NUM and REM should be the same size as DEN.)

Since we can't easily time a single division routine, we'll call the routine 32,768 times - just for the test. We can use the jiffy clock for the benchmark timing.

The benchmark program ("00test") is written in BASIC 7.0 for the 128 (program 0). It inputs the two numbers, zeroes out the clock, SYSes to the routine, and prints the jiffy clock value. The routines assemble to $0B00 on the 128. If you own a 64, you'll have to move them somewhere suitable ($C000, for example) and modify the BASIC program by removing the FAST statement in line 20.

Just to be fair, we should write a program ("01null") that does nothing but clear the carry (CLC) and return from the subroutine (RTS). The clock will report the overhead required for doing things like checking for a division by zero error.

All of the example programs contain the 32K loop and the error-checking for the presence of zero in DEN. If you're working from the listings, type in Program "01null" and for the others, start at line 570 MAIN = * (in the "Buddy" assembler for the 128, the character "*" marks the current program counter; the same is true for "PAL" and "LADS" and some other assemblers). Lines 580+ are different in the example programs.

**Subtract a lot**

Program "02subt" subtracts a lot. The algorithm breaks down to a few simple steps:

1. Begin loop.
2. If NUM < DEN, exit loop.
3. Else NUM = NUM - DEN.
4. Increment counter and repeat the loop.
5. RESult = counter. REMainder = value left in NUM.

When we check the benchmarks, the subtraction alogrithm works reasonably well when the result is small. To evaluate 6/2, the program subtracts two a total of three times. However, the larger the result, the longer the delay. If you divide 60,000/2, the algorithm must subtract 30,000 times. The answer is accurate, but you have to wait quite a while for it (especially if the program repeats the calculation 32K times).

It's also not very satisfying. Subtracting one number from another doesn't seem very elegant. There must be a better way.

**Shift to the right**

Program "03shif" provides a second solution. It's faster than blazes, but (unfortunately) it only works when the divisor DEN is an even multiple of two.

The theory is relatively easy to understand. Suppose your computer worked in decimal (which it doesn't). Suppose the number 130,000 was stored in memory as the numerals "130000" (also a false statement, but we can pretend). Suppose once more that you could tell the numerals to move once to the right (shifting a zero into the high position). Shift 130000 to the right and you get 013000. Shift even more and you get 001300, 000130, 000013, 000001, 000000, and so on.

As you can see, in base ten (decimal), shifting to the right is the same as dividing by ten. But computers store values in binary (base two). Shifting a byte (or series of bytes) right is the same as dividing by two:

| Binary | Decimal |
|---|---|
| 0000 0010 1100 0000 | 704 |
| 0000 0001 0110 0000 | 352 |
| 0000 0000 1011 0000 | 176 |

When the 1's and 0's shift to the right, they fall off the edge of the byte into the carry bit. From there, we can catch the bit (shifting right again into a REMainder area). The shift algorithm for division looks like this:

1. Set a counter to 16 (the size in bits of DEN).
2. Shift DEN right one bit (LSR the high byte and ROR the intermediate/low bytes). If the carry is set, skip ahead to step 6.
3. Shift NUM right one bit (LSR the high byte first, then ROR any intermediate or low bytes).
4. Shift the remaining leftover bit into the high bit of REMainder (using ROR on the high and low bytes).
5. Decrement the counter and branch to 2 if it's not zero yet.
6. The shifting loop has ended and if DEN was a multiple of 2, no more bits should be turned on. Even multiples of two contain only one bit (00000001, 00000010, 00000100, and so on). When the single bit rotated out of DEN, it should have left behind no other 1 bits. If any bits are still on, there's an error, so we set the carry. Else, clear the carry and RTS.

As mentioned above, this trick only works when you know in advance that you have a divisor that's evenly divisible by two. A better routine would be faster than subtracting and handle all possible numbers.

### Calling the ROM routine

Since BASIC handles division reasonably well, we could dig out the appropriate memory map ("Mapping the 64" or "Mapping the 128" or one of the Abacus ROM disassemblies) to find out which ROM routine performs division, and also find out what it requires of our calling program.

The routine happens to reside at $8B4C on the 128 and $BB12 on the 64. However, the FDIVT routine, as it's called, requires two floating-point numbers in FAC1 and FAC2, so we have to do some setup work. We've got integers. They must be converted to floating point.

Program "04floa" starts by loading the high byte of NUM into the accumulator (.A) and the low byte into the Y register (.Y). Then it calls GIVAYF ($AF03 on the 128, $B391 on the 64), which transforms the integer into an official floating-point variable suitable for use by BASIC ROM routines. The number now resides in FAC1. Next, we call MOVEF ($8C3B or $BC0F), which moves a number from FAC1 to FAC2. The final preparatory step is to call GIVAYF again, this time to translate DEN to floating-point format.

The result is a floating-point value in FAC1. To get that back to integer form, jump indirectly through the pointer at ADRAY1 ($117A on the 128, $0003 on the 64). A word of warning about location 3 on the 64: No ROM routines ever jump through 3, so some machine language programmers consider it an available zero-page location. If you've run other assembly language programs before you try this one, you may want to turn your 64 off and on, to make sure the pointer value is correct.

One more thing: The instruction JMP ($117A) is how you jump indirectly to the two-byte address contained at location $117A. There's no equivalent JSR instruction. At this point in the routine, there are a couple more things to do; so it's necessary to place the indirect jump at the end of the program and then JSR to the JMP. That way, the RTS at the end of the conversion routine returns control to our program rather than ending our program.

To summarize the routine that calls ROM subroutines:

1. LDA with the high byte of NUM and LDY with the low byte.
2. JSR GIVAYF
3. JSR MOVEF
4. LDA and LDY with the bytes from DEN.
5. JSR GIVAYF
6. JSR FDIVT
7. JMP (ADRAY1)

There are a few good things to say about calling the ROM division routine. It's fairly short, which can be a factor if you're writing long programs. It takes a fairly constant time (unlike Program "02subt," which varies in time according to the size of the result).

Are there bad things to say? Yes. My own feeling is that the BASIC ROM routines are there for the benefit of BASIC programmers. If you want to call BASIC routines, you might as well write the program in BASIC. In a program that needs to do a lot of division, there's no real reason to rely on the ROM routines if you're writing in assembly language.

A second problem is that you get the result, but not the remainder (although if you wanted to, you could call another series of BASIC routines to get the remainder).

A third problem is that you lose some accuracy when you perform floating-point calculations (try this, for an example: for x = 1 to 5 step 0.1: print x: next).

### The answer: binary division

Program "05divi" does it the right way. Before looking at binary numbers, let's review what happens in decimal division. Say you want to divide 00043299 (which we'll call NUM) by 00000492 (DEN). We will pretend that all decimal values are eight-digit numerals for this gedanken experiment.

Shift NUM to the left and the leftmost numeral will fall out. We'll put it into REM (the remainder). After one shift, the numbers look like this:

| RES | REM | NUM | DEN |
|---|---|---|---|
| zzzzzzzz | zzzzzzz0 | 0043299z | 00000492 |

The z's represent zeros that are there by default. How many times does DEN (492) fit into REM (0)? Zero times. Put that result into RES. This process occurs a few more times before something interesting finally happens:

| RES | REM | NUM | DEN |
|---|---|---|---|
| zz000000 | z0004329 | 9zzzzzzz | 00000492 |

You may have noticed that after seven attempts, we've finally got a number in REM that's bigger than DEN. At this point, we divide DEN into REM. How many times does it fit? Maybe 7, maybe 8, maybe 9. In base ten, you have to keep guessing until you have the right answer, which turns out to be 8. Move that number into RES, multiply 492 by 8 and subtract from REM:

| RES | REM | NUM | DEN |
|---|---|---|---|
| z0000008 | 00003939 | zzzzzzzz | 00000492 |

In the final pass, we calculate that DEN fits into REM 8 times (again). Shift that into RES. Multiply DEN by 8 and subtract from REM. Here's the final answer:

| RES | REM |
|---|---|
| 00000088 | 00000003 |

The result is 88 with a remainder of 3. If you try some base ten division problems, you can understand the basic steps.

**No guess, no mess**

The step where you examine 4329 and ask "How many times does 492 divide 4329?" is the step that stops a lot of programmers. It just seems so messy. You can't really figure out the answer by looking at the numbers (at least I can't). Most of us learn division in elementary school and that rule about guessing is stuck in the back of our minds.

If you've done a significant amount of assembly language programming, but you've never written a division routine, you might be anticipating a loop that checks a bunch of numbers until it gets the right one. You'd be wrong.

We're working in binary now and that simplifies everything. Suppose we're dividing 01101110 by 00000101. Don't worry about what the decimal equivalents are, think in terms of ones and zeros. The divisor 101 obviously won't fit into 0, 01, or 011. In the fourth pass, we get this situation:

| RES | REM | NUM | DEN |
|---|---|---|---|
| zzzzz000 | zzzz0110 | 1110zzzz | 00000101 |

REM is bigger than DEN, so DEN must fit into REM at least once. We'll guess that 110/101 is 1 or 2 or 3 or something else. But this is base two and the only numerals available are 0 and 1. We're shifting the variables one bit at a time, so the answer has to be 1 (if it fit twice, we would have gotten the answer on the previous pass through the loop). There's no guessing at all. The answer is either/or. Either DEN fits into REM or it doesn't. Either the next digit in RES is a 1 or a 0.

It gets even simpler. In the base ten example, we got the digit 8 as one of the intermediate results. We had to multiply 8 by 492 (to get 3936) and subtract 3936 from the number in REM.

We've already decided that the only two digits available in binary are 0 and 1. We'll have to multiply DEN by 0 (or multiply by 1) and then subtract that value from REM. Do we need a separate subroutine to multiply two numbers? Not really.

One of the universal rules about multiplying is "Any number times zero is zero." Another is "Any number times one is the number." A rule about subtracting is "Any number minus zero is the number." In all number bases, these rules are always true. They're not even unwritten.

That means that if DEN fits into REM, you shift a 1 into RES (no guessing) and you subtract (1 * DEN), which is the same as DEN (no multiplying). If DEN doesn't fit, shift a zero into RES (still no guessing) and subtract (0 * DEN), which means subtract 0, which means no math at all (no multiplying or subtracting).

**If you CMP, you get a useful bit**

There's one more 6502-processor rule that makes life even simpler for the programmer writing a division routine.

Recall that we want to test REM and DEN to see if DEN fits into REM. Suppose that we use the two instructions LDA REM: CMP DEN. If REM is greater than or equal to DEN, the carry bit is set. If REM is smaller, the carry is clear. The carry is either 1 or 0, and that's the next digit we want to shift left into RES.

In addition, if the carry is 0, we'll leave REM alone. If the carry is 1, we need to subtract (REM - DEN) and store the new value in REM. An advocate of structured programming might scream at the sight of the following code, but we're going do it anyway:

```
         lda rem
         cmp den
         bcs subtr
   xx    rol res
         rts
 subtr   jsr xx
         sec
         lda rem
         sbc den
         sta rem
         rts
```

If the carry is clear, the digit is zero. The program drops through to the code at xx, which rotates the carry left into RES and then returns. If the carry is set, the program branches ahead to SUBTR, which immediately jumps to the subroutine at xx. The carry is still set, so it rotates left into RES. In this case, the ReTurn from Subroutine (RTS) doesn't make us exit the loop, it sends the program back to SUBTR. The SEt Carry (SEC) instruction is necessary because the carry flag was likely zeroed out by the ROL instruction. We subtract DEN from REM and exit.

The general algorithm looks like this:

1. Set a counter to 16 (the size in bits of NUM)
2. Begin the loop.
3. Shift all bits of NUM to the left, starting with the least significant byte.
4. Shift the leftover bit (in the carry) into REM.
5. If REM < DEN then shift the carry flag (0) into RES.
6. Else REM >= DEN, so shift the carry flag (1) into RES and subtract. REM = REM - DEN.
7. Decrement the counter. If it's not zero, repeat the loop.

**Which is the fastest?**

The table below lists the times (in jiffies, where a jiffy is 1/60th second) for five routines running on a Commodore 128 in FAST (2 MHz) mode. "01null" is the program that doesn't do anything. "02subt" subtracts repeatedly. "03shif" shifts bytes to the left. "04floa" calls BASIC ROM routines. "05divi" uses the binary division routine.

NUM and DEN are the numerator and denominator. RES and REM are the result and remainder.

**Table 1:** *Evaluating the various algorithms for speed.*

| NUM | DEN | RES | REM | 01 | 02 | 03 | 04 | 05 |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 116 | 216 | 439 | 3267 | 1387 |
| 100 | 1 | 100 | 0 | 115 | 6115 | 439 | 2897 | 1463 |
| 6 | 2 | 3 | 0 | 115 | 335 | 466 | 3176 | 1425 |
| 51 | 2 | 25 | 1 | 115 | 1646 | 466 | 3044 | 1463 |
| 60100 | 2 | 30050 | 0 | 115 | 1419968 | 466 | 3268 | 1653 |
| 1 | 64 | 0 | 1 | 115 | 156 | 599 | 3021 | 1349 |
| 33333 | 64 | 520 | 53 | 115 | 24784 | 600 | 3414 | 1425 |
| 10000 | 5000 | 2 | 0 | 115 | 251 | -- | 2853 | 1233 |
| 60100 | 5000 | 12 | 100 | 115 | 723 | -- | 3797 | 1273 |
| 60100 | 30000 | 2 | 100 | 115 | 251 | -- | 3444 | 1233 |

Ignore column 01. The first program doesn't divide at all. Column 02 is the subtraction algorithm. The answer to "Do you subtract a lot?" is clearly "Not if you can help it." When the result is a large number, subtracting is an unattractive alternative (32768 repetitions of 60100 divided by 2 requires 1.4 million jiffies, approximately one hour, or 40 seconds per calculation). Column 03 looks good, but the algorithm only works if the divisor is an even multiple of two. Columns 04 and 05 contain reasonably stable numbers.

### 00test: *BASIC 7.0 benchmark program*

```
PP  10 rem scratch"00test":dsave"00test"
PH  20 fast
KI  25 for i = 1 to 10
OM  30 readj,k:l=int(j/k):printl,j-l*k,
EH  40 jh = int(j/256):jl = j-jh*256
JI  50 kh = int(k/256):kl = k-kh*256
FO  60 poke2816,jl:poke2817,jh:poke2818,kl:poke2819,kh
BK  70 ti$="000000"
FJ  80 sys 2830
FI  90 print ti;"jiffies"
CB  100 forj=2820to2822step2:printpeek(j)+256*peek(j+1),:next:print
CH  110 next
HO  120 data 1,1,100,1,6,2,51,2,60100,2
FO  130 data 1,64,33333,64,10000,5000,60100,5000,60100,30000
```

### Common code: *All five ML programs begin with this code.*

```
MI  100 sys 4000
PN  110             .bank 15
KD  120             .org $0b00
GC  130             .mem
GK  140 n1          .word 0  ; numerator passed
AI  150 n2          .word 0  ; denominator passed
MD  160 res         .word 0  ; result
FM  170 rem         .word 0  ; remainder
MH  180 num         .word 0  ; numerator
KM  190 den         .word 0  ; denominator
EA  200 count1      .byte 0
BB  210 count2      .byte 0
CF  220 ;
BN  230             lda #0
EG  240             sta count1
NF  250             lda #128
KH  260             sta count2
KC  270 doit32k     dec count1
KF  280             bne iterate
FF  290             dec count2
OG  300             bne iterate
CC  310             rts
FO  320 iterate     jsr routine
IC  330             bcc doit32k
AE  340             rts
EN  350 ;
CM  360 ; if routine set the carry, an error occurred
IO  370 ;
CK  380 routine     = *        ; check for divide by zero error
AK  390             lda n2
EA  400             ora n2+1
MC  410             bne init
FC  420             sec
MP  430             rts        ; zero out res and rem
IL  440
BK  450 init        lda #0
HE  460             sta res
KC  470             sta res+1
FB  480             sta rem
IA  490             sta rem+1
LL  500 ; copy n1 to num and n2 to den
PE  510             ldy #3
JB  520 loop        lda n1,y
AK  530             sta num,y
HN  540             dey
BO  550             bpl loop
GK  560 ;
BH  570 main        = *
```

### 01null: *Append to "Common code" to create "01null"*

```
BH  570 main        = *
KL  580             clc
KD  590             rts
```

### 02Subt: *Append to "Common code" to create "02subt"*

```
BH  570 main        = *
ND  580             lda den+1
GF  590             cmp num+1    ;  double-compare high bytes
CD  600             bcc subtract; if den < num, continue
KA  610             beq lowbyte ; else if equal, check the low byte
CO  620 ;
GN  630 quit        lda num
FL  640             sta rem
KK  650             lda num+1
CL  660             sta rem+1
EB  670             clc
GF  680             rts          ; else den > num, so we quit
IC  690 ;
PF  700 lowbyte     lda num
LC  710             cmp den
FE  720             bcc quit     ; if num < den then quit
CA  730 ; else drop through to subtract
NC  740 subtract    sec
FD  750             lda num
IC  760             sbc den
HI  770             sta num
MC  780             lda num+1
PB  790             sbc den+1
OH  800             sta num+1
HI  810             inc res
IJ  820             bne main
EH  830             inc res+1
FO  840             jmp main
```

### 03shif: *Append to "Common code" to create "03shif"*

```
BH  570 main        = *
CI  580 numbits     = 16         ; number of bits in denominator
AC  590             ldx #numbits
OB  600 mloop       lsr den+1
HA  610             ror den
LJ  620             bcs cleanup
HA  630             lsr num+1
LF  640             ror num
HA  650             ror rem+1
EI  660             ror rem
FF  670             dex
II  680             bne mloop
EN  690 cleanup     lda den
CJ  700             ora den+1
```

```
OM  710          beq itsok
BF  720          sec
GM  730          rts
KF  740 ;
HD  750 ; if any bits are still set, den isn't a multiple of 2 (error)
OG  760 ;
GK  770 itsok    lsr rem+1
MP  780          ror rem
NM  790          dex
DF  800          bne itsok   ; fix the remainder
BH  810          lda num
PK  820          sta res
OF  830          lda num+1
MJ  840          sta res+1
IM  850          clc
LJ  860          rts         ;  clear carry means success
```

**04floa:** *Append to "Common code" to create "04floa"*

```
BH  570 main     = *
FP  580 fdivt    = $8b4c     ; $bb12 on the 64
LF  590 movef    = $8c3b     ; $bc0f on the 64
GD  600 givayf   = $af03     ; $b391 on the 64
KE  610 adrayl   = $117a     ; $0003 on the 64
MI  620          lda num+1
BK  630          ldy num     ; get the high and low byte
HP  640          jsr givayf  ; convert to floating-point in facl
AI  650          jsr movef   ; move to fac2
NI  660          lda den+1
NG  670          ldy den     ; high and low again
BL  680          jsr givayf  ; convert (facl again)
GP  690          jsr fdivt   ; call the rom division routine
IA  700          jsr convert ; converts facl to an integer
KB  710          sta res+1
AC  720          sty res     ; in .a and .y
AF  730          clc
AN  740          rts
HI  750 convert  jmp (adrayl)
```

**05divi:** *Append to "Common code" to create "05divi"*

```
BH  570 main     = *
CD  580 numbits  = 16        ; number of bits in num
AC  590          ldx #numbits
HN  600 mainloop jsr divide
JB  610          dex
HO  620          bne mainloop
MO  630          clc
MG  640          rts
PF  650 divide   asl num
NA  660          rol num+1
OO  670          rol rem
BO  680          rol rem+1
CJ  690          lda rem+1
MG  700          cmp den+1   ; compare the high bytes
HD  710          beq notsure
ON  720 ; if equal, can't tell which is bigger
BM  730          bcs doublesub
PH  740 ; if rem is bigger, subtract den
DL  750 answerbit rol res
HG  760          rol res+1
OO  770          rts
JC  780 ; get the carry into the result
CO  790 notsure  lda rem
FI  800          cmp den
BH  810          bcc answerbit ; another zero bit?
PM  820 doublesub jsr answerbit
IF  830 ; even if carry is set, put it into res
JM  840          sec
JE  850          lda rem
MI  860          sbc den
LJ  870          sta rem
AF  880          lda rem+1
DI  890          sbc den+1
CK  900          sta rem+1
KH  910          rts
```

# Fast Graphics Primitives

*Assembler routines written for flat-out performance*

**by Robert Huehn**

Any attempt to produce real-time graphics animation on any computer will inevitably be limited by its processing speed. The C64, with a 1 Mhz 6502-series microprocessor, imposes hard limits on the complexity of graphics that can be accomplished within a reasonable length of time. You need to decide which is most important - speed or complexity - because it may be impossible to achieve both at once. (For a good example of the tradeoff, consider subLOGIC's flight simulators.) Obviously, fast graphics routines are necessary for best results.

This article deals specifically with optimizing hi-res graphics for the C64. The discussion and source that follow illustrate several general principles which can probably be applied to other situations or other machines.

### No BASIC here

First, let's examine carefully the general scenario where very fast graphics routines are required. Such a program will have a set of routines to calculate the graphics data, and a set of graphics routines that draw the graphics data to the screen. If the calculation part of the process consumes the largest portion of processor time, then faster graphics primitive routines will not serve to any great advantage. This rules out BASIC as the your programming language in most cases. In this case, it can be assumed that your calculation routines are written in machine language, and that most processor time is used simply to do the drawing. You need fast graphics primitives.

Based on the above assumptions, the routines that follow were written to provide flat-out performance. They expect that BASIC ROM is swapped out; this provides space for the bitmap and lots of zero-page storage. As they stand now, the routines are not really flexible and will require some modification if you want to locate the bitmap at another address, or change drawing modes, or whatever. But this is one key to obtaining good performance: by restricting routines to simple goals, they can make up for the loss of versatility with increased speed.

### Unrolled loops

Animation is normally accomplished by quickly displaying successive frames on the screen in order to create the illusion of motion. This involves drawing the screen, clearing it, redrawing, and so on. Usually a double-buffering system is used to reduce flicker, where drawing is done to a second screen area in memory and displayed once the frame is complete.

Consider possible ways to clear the 8000 bytes that the bitmap screen consists of. The standard method would involve a nested loop with two levels, containing an indirect indexed 'STA (address),y'. This addressing mode requires six clock cycles; add the overhead from the looping instructions, and you'll find this method requires an average of eleven cycles to clear each byte. That overhead can be cut down by partially unrolling the loop: by including more STA instructions inside it, the loop can be performed fewer times.

A loop to fill 8000 bytes containing 32 STAs needs to be executed only 250 times. This requires only one index register, simplifying control of the loop. Also, indexed absolute addressing can be used; using five cycles instead of six for each STA. The *bmclr* routine in the source code works this way, requiring an average of just 5.2 cycles per byte! A disadvantage to unrolled loops is the increase in the size of the code.

### Fast plot

Common sense tells you that one of the best places for optimization is within loops. These graphics routines were created with that in mind; they will be the innermost level of any graphics program, and extra cycles would be costly. The most crucial of all is the plot function, which is central to almost all other routines.

*The plot routine takes 70 cycles to execute (including the RTS)*

The plot routine will be called with an x coordinate between 0 and 319 and a y coordinate between 0 and 199, with the origin at the top left corner of the screen. These coordinate values are passed in zero page and are assumed to be legal values. The format of the bitmap requires some calculation to find the address and bit position of the pixel. The *Commodore 64 Programmer's Reference Guide* suggests:

```
row = int (y/8)
char = int (x/8)
line = y and 7
bit = 7-(x and 7)
byte = base + row*320 + char*8 + line
poke byte,peek(byte) or 2^bit
```

However, the byte calculation can be reduced to:

```
byte = base + 40*(y and 248) + (x and 504) + line
```

...which works faster in machine language.

A powerful yet simple technique is to use look-up tables instead of a specialized section of code. For example, the bit calculation is done by indexing into a table with the lower three bits of x, to produce a bit mask. Notice the multiply by 40 - you don't really want to do multiplication in such a critical routine. Again, indexing a table of values is faster. The top five bits of y are shifted right twice, to index into a table of words pointing to the beginning of each row. To save more time, the base address of the screen has already been added in to this table.

There is a subtle trick in the plot source code, used to avoid two extra 'AND #248' instructions. Right after each coordinate is masked with 'AND #7' to produce the lower three bits, an EOR with the same coordinate produces the top five bits!

The plot routine seems to be fully optimized now - at least improvements are not obvious. It takes 70 cycles to execute (including the RTS.)

**Fast draw**

The speed of a line draw depends on the algorithm used. The standard algorithm comes from *Principles of Interactive Computer Graphics* by Newman and Sproull; an improvement by Mike Higgins (p. 414, August 1981 *BYTE*) requires only addition and subtraction. This information is from Larry Isaacs' column in the April 1984 issue of *COMPUTE!* (p. 128).

It works something like this:

```
Draw (x1, y1, x2, y2)

   xinc = POS: yinc = POS
   dx = x2 - x1
   if dx < 0
      xinc = NEG: dx = abs(dx)
   dy = y2 - y1
   if dy < 0
      yinc = NEG: dx = abs(dy)
   x = x1: y = y1
   Plot (x, y)
   if dx > dy
      r = dx/2
      for c = 1 to dx
         x = x + xinc
         r = r + dy
         if r >= dx
            y = y + yinc
            r = r - dx
         Plot (x, y)
      next
   else
      r = dy/2
      for c = 1 to dy
         y = y + yinc
         r = r + dx
         if r >= dy
            x = x + xinc
            r = r - dy
         Plot (x, y)
      next
```

If this routine calls the plot function each time through the loop, then it will take 70 cycles, plus the time for the rest of the loop, for each pixel on the line. But the fast draw routine avoids unnecessary calculation by updating the byte address and bit location directly. A modified plot is included to set up these values and to plot the first pixel. A test of the fast draw, drawing a line diagonally across the screen 256 times, took seven seconds to execute. That is 12000 pixels per second, or around 85 cycles per pixel! Lines which are closer to the horizontal or vertical axes will be even faster - a horizontal line uses 71 cycles per pixel.

**Take the challenge**

Feel free to attempt to improve the performance of these routines: should you succeed, I'm sure other readers would like to see your results - so please send a letter decribing any improvements. Fast graphics primitives are the first step to really impressive graphics effects.

*Symass-compatible source code follows on the next page containing the fast graphics routines and a demo routine to show off their speed. Acknowledgement: Thanks to Glen MacKinnon for some really great ideas, including the EOR trick.*

```
KD 100 sys700 ;run assembler
EO 110 ;
II 120 ;  <<<  graphics  v1.0  >>>
PO 130 ; copyright 1988 by robert huehn
FK 140 ; high speed graphic routines
BG 150 ; jan 1988
GB 160 ;
NA 170 ;zpage pseudo registers
KM 180 r0 =$02
NN 190 r1 =$04
AP 200 r2 =$06
DA 210 r3 =$08
LC 220 r4 =$0a
OD 230 r5 =$0c
BF 240 r6 =$0e
BC 250 r7 =$10
ED 260 r8 =$12
HE 270 r9 =$14
OI 280 ;
JA 290 *=$9000
DN 295 jmp demo
KG 300 ;jump table
MG 310 bmon jmp ibmon
EF 320 bmoff jmp ibmoff
NM 330 bmclr jmp ibmclr
EE 340 txclr jmp itxclr
KI 350 plot jmp iplot
NC 360 draw jmp xdraw
IO 370 ;
GH 380 bitab =* ;pixel masks
OI 390 .byte 128,64,32,16,8,4,2,1
NG 400 lotab =* ;base addresses
MO 410 hitab =*+1
CF 420 .word $a000,$a140,$a280,$a3c0
IJ 430 .word $a500,$a640,$a780,$a8c0
CD 440 .word $aa00,$ab40,$ac80,$adc0
FM 450 .word $af00,$b040,$b180,$b2c0
CL 460 .word $b400,$b540,$b680,$b7c0
AD 470 .word $b900,$ba40,$bb80,$bcc0
HP 480 .word $be00
AG 490 ;
OI 500 ;turn on bit map at $a000
DI 510 ibmon =*
IN 520 lda $dd00
MP 530 and #$30
CO 540 ora #$01
ED 550 sta $dd00
EK 560 lda #$3b
BB 570 sta $d011
AJ 580 lda #$38
MC 590 sta $d018
EE 600 rts
IN 610 ;
MB 620 ;back to normal text
DC 630 ibmoff =*
AF 640 lda $dd00
EH 650 and #$30
AG 660 ora #$03
MK 670 sta $dd00
GB 680 lda #$1b
JI 690 sta $d011
GP 700 lda #$15
EK 710 sta $d018
ML 720 rts
AF 730 ;
GO 740 ;clear bit map $a000-bf40
OH 750 ibmclr =*
DO 760 lda #0
MK 770 ldx #250
DB 780 cl1 sta $9fff,x
DK 790 sta $a0f9,x
KK 800 sta $a1f3,x
EM 810 sta $a2ed,x
EM 820 sta $a3e7,x
LM 830 sta $a4e1,x
FO 840 sta $a5db,x
FO 850 sta $a6d5,x
PP 860 sta $a7cf,x
PP 870 sta $a8c9,x
GA 880 sta $a9c3,x
FD 890 sta $aabd,x
FD 900 sta $abb7,x
MD 910 sta $acb1,x
GF 920 sta $adab,x
GF 930 sta $aea5,x
EF 940 sta $af9f,x
BB 950 sta $b099,x
IB 960 sta $b193,x
CD 970 sta $b28d,x
CD 980 sta $b387,x
JD 990 sta $b481,x
DF 1000 sta $b57b,x
DF 1010 sta $b675,x
NG 1020 sta $b76f,x
NG 1030 sta $b869,x
EH 1040 sta $b963,x
DK 1050 sta $ba5d,x
DK 1060 sta $bb57,x
KK 1070 sta $bc51,x
EM 1080 sta $bd4b,x
EM 1090 sta $be45,x
DA 1100 dex
GE 1110 bne cl1
ME 1120 rts
AO 1130 ;
LF 1140 ;set bit map colour at $8c00
AH 1150 itxclr =*
JD 1160 lda #$bf
MD 1170 ldx #250
OC 1180 col1 sta $8bff,x
KF 1190 sta $8cf9,x
BG 1200 sta $8df3,x
LH 1210 sta $8eed,x
LH 1220 dex
JN 1230 bne col1
EM 1240 rts
IF 1250 ;
GB 1260 ;fast line draw
BF 1270 idraw =*
MH 1280 ;passed:
HM 1290 x1 =r0
FN 1300 y1 =r1
CO 1310 x2 =r2
AP 1320 y2 =r3
LD 1330 ;altered:
EE 1340 dx =r4    ;delta x
EF 1350 dy =r5    ;delta y
AA 1360 xi =r5+1  ;l/r flag
IH 1370 yi =r6    ;u/d flag
KA 1380 base =r7  ;base of pixel addr
NK 1390 m =r6+1   ;pixel mask
MP 1400 c =r8     ;count
DD 1410 r =r9     ;
HI 1420 ldx #0    ;xinc=right
LI 1430 ldy #0    ;yinc=down
DB 1440 lda x2    ;calculate dx=x2-x1
LC 1450 sec
GO 1460 sbc x1
IF 1470 sta dx
OK 1480 lda x2+1
GM 1490 sbc x1+1
ID 1500 sta dx+1
KP 1510 bcs dr1
JN 1520 dex          ; dx<0, xinc=left
HO 1530 lda #1
AH 1540 sbc dx
IK 1550 sta dx
DA 1560 lda #0
AF 1570 sbc dx+1
II 1580 sta dx+1
MK 1590 dr1 lda y2     ;dy=y2-y1
BM 1600 sec
NH 1610 sbc y1
LG 1620 bcs dr2
IP 1630 dey          ;dy<0, yinc=up
PD 1640 eor #$ff     ;dy=abs(dy)
BF 1650 adc #1
JO 1660 dr2 sta dy
CH 1670 stx xi
BI 1680 sty yi
JF 1690 lda y1       ;plot (x1,y1)
HE 1700 and #7
NH 1710 tay
HO 1720 eor y1
LJ 1730 lsr
FK 1740 lsr
BK 1750 tax
CA 1760 lda x1
HP 1770 and #$f8
KN 1780 adc lotab,x
PH 1790 sta base     ;save base
MO 1800 lda hitab,x
IO 1810 adc x1+1
PJ 1820 sta base+1
IE 1830 lda x1
DN 1840 and #7
FA 1850 tax
CC 1860 lda bitab,x
PG 1870 sta m        ;save mask
IH 1880 ora (base),y
CE 1890 sta (base),y
KI 1900 lda dx+1
LL 1910 bne dri
NG 1920 lda dx       ; (dx>=dy)
NC 1930 cmp dy
AP 1940 bcs dri
AE 1950 jmp drii
AG 1960 dri =*       ;case i -1<slope<1
AN 1970 lda dx+1
LE 1980 sta c+1      ;c=dx
PJ 1990 lsr
OI 2000 sta r+1      ;r=dx/2
GD 2010 lda dx
NM 2020 sta c
CK 2030 ror
AP 2040 sta r
NK 2050 lda c
HE 2060 ora c+1
AO 2070 beq dr9      ;if single point
KK 2080 dr3 lda xi
ND 2090 bmi dr4
NL 2100 lsr m        ;right
OB 2110 bcc dr5
AD 2120 ror m
LI 2130 lda base
JE 2140 adc #8
NN 2150 sta base
AF 2160 bcc dr5
PN 2170 inc base+1
NI 2180 bne dr5
JO 2190 dr4 asl m    ;left
IH 2200 bcc dr5
AL 2210 rol m
FO 2220 lda base
PL 2230 sbc #7
HD 2240 sta base
```

```
KO  2250 bcs dr5
EB  2260 dec base+1
JI  2270 dr5 lda r     ;r=r+dy
OF  2280 clc
CE  2290 adc dy
EP  2300 sta r
JO  2310 bcc dr6
BN  2320 inc r+1
GB  2330 dr6 sec
AJ  2340 sbc dx
JP  2350 tax
JN  2360 lda r+1
AH  2370 sbc dx+1
FD  2380 bcc dr8
MM  2390 stx r         ;r>=dx,
HH  2400 sta r+1       ;r=r-dx
NL  2410 lda yi
AJ  2420 bmi dr7
IL  2430 iny ;down
FP  2440 cpy #8
LH  2450 bcc dr8
HO  2460 ldy #0
PN  2470 lda base
GC  2480 adc #$3f
BD  2490 sta base
JA  2500 lda base+1
NK  2510 adc #1
MI  2520 bcc dr18
GP  2530 dr7 dey ;up
AC  2540 bpl dr8
PE  2550 ldy #7
JD  2560 lda base
JE  2570 sbc #$40
LI  2580 sta base
DG  2590 lda base+1
FC  2600 sbc #1
PC  2610 dr18 sta base+1
OC  2620 dr8 lda (base),y
NJ  2630 ora m
CD  2640 sta (base),y ;plot (x,y)
AA  2650 dec c
HG  2660 bne dr3
LP  2670 dec c+1
OP  2680 beq dr3       ;next
HN  2690 dr9 rts
ID  2700 drii =*       ; -1>slope>1
EP  2710 lda dy
KO  2720 beq dr15      ;single point
AJ  2730 sta c         ;c=dy
NI  2740 lsr
PK  2750 sta r         ;r=dy/2
NI  2760 dr10 lda yi
AK  2770 bmi dr11
GB  2780 iny ;down
DF  2790 cpy #8
MI  2800 bcc dr12
FE  2810 ldy #0
ND  2820 lda base
EI  2830 adc #$3f
PI  2840 sta base
HG  2850 lda base+1
LA  2860 adc #1
OO  2870 bcc dr19
DE  2880 dr11 dey      ;up
BD  2890 bpl dr12
NK  2900 ldy #7
PN  2910 sec
BK  2920 lda base
BL  2930 sbc #$40
DP  2940 sta base
LM  2950 lda base+1
NI  2960 sbc #1
IJ  2970 dr19 sta base+1
PF  2980 dr12 ldx #0
OG  2990 lda r         ;r=r+dx
OC  3000 clc
AB  3010 adc dx
EM  3020 sta r
CM  3030 bcs dr16
ML  3040 inx
LG  3050 sec
KN  3060 dr16 sbc dy
OO  3070 bcs dr17
PL  3080 dex
EP  3090 beq dr14
AA  3100 dr17 sta r    ;r>=dy, r=r-dy
IH  3110 lda xi
GA  3120 bmi dr13
DM  3130 lsr m ;right
IO  3140 bcc dr14
GD  3150 ror m
BJ  3160 lda base
PE  3170 adc #8
DO  3180 sta base
KB  3190 bcc dr14
FO  3200 inc base+1
HF  3210 bne dr14
NF  3220 dr13 asl m    ;left
CE  3230 bcc dr14
GL  3240 rol m
LO  3250 lda base
FM  3260 sbc #7
ND  3270 sta base
EL  3280 bcs dr14
KB  3290 dec base+1
MP  3300 dr14 lda (base),y
FE  3310 ora m
KN  3320 sta (base),y ;plot (x,y)
IK  3330 dec c
IA  3340 bne dr10      ;next
JH  3350 dr15 rts
GJ  3360 ;
II  3370 ;fast plot
DK  3380 iplot =*
KL  3390 ;passed:
LD  3400 xc =r0
JE  3410 yc =r1
FG  3420 ;altered:
IB  3430 ;base =r7
II  3440 ptemp =r6+1
BM  3450 lda yc
HC  3460 and #7
PH  3470 sta ptemp
JN  3480 eor yc
LH  3490 lsr
FI  3500 lsr
BI  3510 tax
EK  3520 lda hitab,x
EM  3530 adc xc+1
HF  3540 sta base+1
CN  3550 lda lotab,x
PF  3560 sta base
ID  3570 lda xc
PJ  3580 and #7
BN  3590 tax
NE  3600 eor xc
PL  3610 adc ptemp
DP  3620 tay
AN  3630 lda (base),y
LP  3640 ora bitab,x
CC  3650 sta (base),y
ID  3660 rts
MM  3670 ;
GM  3680 ; show-off demo
KM  5000 demo =*
GN  5010 lda #$36
DH  5020 sta 1
KA  5030 jsr bmon
FD  5040 jsr bmclr
HG  5050 jsr txclr
PK  5060 lda #0
OC  5070 sta x1
KP  5080 sta x1+1
DE  5090 sta y1
AB  5100 sta x2+1
MI  5110 lda #$9f
CG  5120 sta x2
LE  5130 lda #$63
HH  5140 sta y2
EJ  5150 ;
OH  5160 lp1 =*
OA  5170 jsr draw
KL  5180 ldx x1
CC  5190 inx
MA  5200 stx x1
OF  5210 bne lp2
IG  5220 inc x1+1
PG  5230 bne lp1
OO  5240 ;
MN  5250 lp2 =*
OH  5260 cpx #$3f
HJ  5270 bne lp1
AO  5280 ldx x1+1
AM  5290 beq lp1
KC  5300 ;
MB  5310 lp3 =*
EK  5320 jsr draw
BF  5330 ldx y1
IL  5340 inx
DK  5350 stx y1
GN  5360 cpx #$c7
BA  5370 bne lp3
AO  5380 jsr draw
EI  5390 ;
KH  5400 lp4 =*
PD  5410 dec x1
IA  5420 jsr draw
EL  5430 ldx x1
KE  5440 bne lp4
JC  5450 dec x1+1
DH  5460 beq lp4
CG  5470 inc x1+1
ON  5480 ;
IN  5490 lp5 =*
IF  5500 jsr draw
EK  5510 dec y1
NJ  5520 bne lp5
AI  5530 lpw lda 197
AH  5540 cmp #60
BC  5550 bne lpw
EL  5560 jsr bmoff
KA  5570 lda #$37
DK  5580 sta 1
CM  5590 rts
GF  5600 ;
DG  5610 xdraw =*
HO  5620 lda #4
MP  5630 inc $fb
NC  5640 bit $fb
EH  5650 bne xd
IA  5660 rts
MJ  5670 ;
AC  5680 xd =*
HB  5690 jmp idraw
```

# Cycle Counting

## *A technique for writing faster code*

**by David Sanner**

Throughout this article, I will be concerned mainly with a single aspect of program analysis: execution speed. The specific focus will be on the technique of "cycle counting" for determining the precise execution speed of a machine language or BASIC program. We'll also look at some of the techniques used to write faster code.

The quest for speed from a computer is unending. One way to speed program execution is to use 'canned' algorithms. For example, sorting algorithms, such as those found in computer science textbooks, have become quite sophisticated. Many of them have been analyzed with mathematical techniques. This allows us to compare their respective performances.

But an in-depth mathematical analysis of a program can be quite complex, and may require skills the general programmer does not possess. Of course, this does not mean one cannot apply simpler methods of analysis. For instance, you might use a stopwatch to measure the time it takes different spreadsheets to recalculate the same data.

But in the case of an ML subroutine, the execution time is often very small - much less than the time it takes to operate a stopwatch. In terms of the computer's speed, however, the code that has just been executed may be very slow, inefficient, and wasteful of valuable processor time. A good example where efficient code is a necessity occurs when computers try to communicate with other devices (such as disk drives). The speed at which they can send data to each other is often staggering - thousands to millions of bits in one second. At this speed the code used to communicate and transfer data must be able to respond quickly, or things will go awry.

Other examples of the need for speed include: changing the screen colour during a horizontal blank interrupt, smooth animation, controlling background music, or "fine-tuning" your code to run as fast as possible. It is clear that for these cases, the human time frame will not do the job. We will have to meet the computer on its own ground.

### The cycles of the 6510

The 6510 inside your C64 is controlled by a clock. The 'ticks' of this clock control how fast the processor can manipulate information (each tick is called a "cycle"). In the C64, a cycle occurs every one millionth of a second (approximately). Thus we say the processor runs at about one megahertz.

The 6510 has 56 op-codes. Each requires a certain number of cycles to execute. The exact number depends on several factors, but is always between two and seven. The first factor, obviously, is which instruction you are using. The second factor is its addressing mode. The last is the 'effective address'.

### Effective addresses

The effective address is the final address the processor will use to access memory. This address is calculated in the processor, by taking into account the addressing mode (indexed, indirect, zero page, etc.) and the base address.

For example, consider the instruction LDA $F000. It requires three cycles to execute, and uses the absolute addressing mode. The effective address of this instruction is simply the address given, $F000. The instruction LDA ($F0),Y, however, uses the indirect indexed addressing mode, which makes calculating the effective address somewhat more involved. It requires six cycles to execute.

To calculate the effective address in this case, the processor grabs the two bytes starting at $F0. It forms the base address using the byte at $F0 as the low byte, and the value in $F1 as the high byte. Then it adds in the contents of the Y register to this base address. This forms the effective address for the indirect indexed addressing mode.

I said that this instruction requires six cycles. If you examine a chart of the 6510 op-codes, however, you will notice a footnote on this instruction. It could be phrased: "If the effective address crosses the base address page, add one cycle to the execution time."

The concept of a page is an important one for calculating cycle times. Briefly, the 6510's address space is divided into 'pages' - groups of 256 bytes. The lowest page is known as page zero - $0000-$00FF. Page one occupies $0100-$01FF, page two from $0200-$02FF, and so on. Note that the page number is simply the high byte of the address.

Regarding the footnote, it's now clear that if the value of the Y register causes the effective address to cross the page pointed to by $F0 and $F1, we must add one cycle.

**Counting cycles by hand**

Let's look at two ML subroutines (see Listing 1 and Listing 2 at the end of this article). Both perform the same task: filling 256 bytes with a value of zero. The starting address is passed with the low byte in X, and the high byte in Y. The fill proceeds from the start address upwards, towards $FFFF.

First, we'll count the cycles needed to execute Listing 1. It may be handy to have a copy of the *Commodore 64 Programmer's Reference Guide* nearby. It contains a table giving the cycle requirements for each op-code.

To simplify our analysis, let's assume that filling will always begin at the start of a page, e.g. $8000. Since we fill from the beginning of a page, there can never be any page crossings when we use STA ($FB),Y or STA $8000,Y. With no page crossings, we don't have to worry about any extra cycles. This would be the "best case" for these subroutines. In the "worst case", we would cross pages on all or most of our STA instructions. This would occur if the start address for filling was at the end of a page, e.g. $80FF.

The first five instructions (up to, but not including the one at *loop1*) take 16 cycles. The main loop (four ops) takes 13 cycles. (Note that we are assuming that the BNE does not cross a page - if it did, that would add one more!). This loop will execute until Y becomes $FF. Since Y is zero when the loop is entered, this makes for 255 passes through the loop. Of these, 254 passes will use 13 cycles (a total of 3302). The last time through, the BNE will fail, using only two cycles to execute. (All branch instructions use three cycles if the branch is taken, and only two if it is not. An extra cycle is needed if the branch address is on a different page than the branch instruction). Adding these 12 cycles for the last loop gives us 3314 for the entire loop. The final instruction, RTS, uses six cycles. Since this is a subroutine, it will be called with a JSR. This takes six cycles as well (note that we are not considering the time it takes to load the X and Y registers with the proper values before the call). All told, 3342 cycles are required for this subroutine to be called, execute, and return.

Listing 2 is two bytes shorter than Listing 1. The first four instructions take 12 cycles. The three instructions in the main loop take only ten cycles. This is where we can gain a lot of speed - with fewer instructions inside a loop, it will execute faster. An obvious point perhaps, but one that can never be stressed enough.

The main loop will be executed 256 times. Of these, 255 loops use ten cycles per loop (2550 cycles). Only nine cycles are used the last time through the loop (because the BNE is not taken), so the loop total is 2559 cycles. The final total of 2583 cycles is a significant improvement (about 30%) over Listing 1.

**A bit about code tweaking**

Sometimes programmers use techniques that exploit 'limitations' or special characteristics of a processor. I call this 'tweaking'. For instance, notice how you can exploit a 'limitation' of register size in Listing 2. When the loop is first entered, Y is zero. After it is incremented, we test it - without comparing it. The BNE is taken if the Z flag is *not* set. There are many instructions that can set or clear the Z flag. The CPY #$30 compares Y to the value of $30 by subtracting - if the result of that subtraction is zero, the Z flag in the status register will be set. This indicates the values are equal.

Listing 2 uses a different method for setting the Z flag, however. Since each register is only eight bits wide (the 'limitation'), the maximum value is $FF. If you execute an INY when Y is $FF, it will 'wrap around' to $00. With the INY instruction, if the processor detects that Y is zero (after it's incremented), it sets the Z status flag.

If we assume that we are at the top of the loop, and Y is $FF, the following will occur: After we store A, we increment Y, thus 'wrapping' Y to zero, and setting the Z flag. The BNE will now fail, because the INY has set the Z flag for us.

Sometimes, shortcuts may violate the rules of 'clean' programming practice, or they may even fail in certain situations. For instance, in Listing 2, notice that indexed absolute addressing is used in the main loop, with Y as the index: STA $0000,Y. This uses fewer cycles than the indexed indirect addressing mode (the STA ($FB),Y) used in Listing 1. It may, however, present some problems.

For the main loop to store A at the proper address, we use the contents of X and Y. Instead of storing them on page zero, as in Listing 1, we store them into the actual instruction area. In other words, we modify the operand of the STA instruction every time the subroutine is called.

This is referred to as 'self-modifying code', since the existing instructions themselves are modified. When you use this technique, errors can occur that are quite difficult to track down. Furthermore, since you actually store data into the program area, you cannot place such a program in ROM.

The location where the start address will be placed is also fixed; if you want to relocate the code, you will have to reassemble it, or modify the first two ops to point to the proper address. Listing 1 avoids this difficulty by using zero page locations.

So, it seems we do not get something for nothing. We can increase the speed by tricks and clever programming, but we may sacrifice the generality of the code. Also, not every subroutine, or line of code needs to run as fast as you can make it. You should ask yourself some questions before you undertake a serious effort. For instance, is the time you will spend squeezing a few more cycles out of your code worth it? You may find yourself spending hours getting the perfect subroutine, but end up neglecting the rest of your code.

The example I gave using Listings 1 and 2 was really quite simple. It took about five to ten minutes to calculate the cycles used for each listing. If you have a larger section of code, the procedure becomes much more complex: not only is there the cycle time of each instruction based on the addressing mode used, but you must also keep careful track of all the status flags, changes in memory locations, etc. Needless to say, it sounds like a job that is more suited for a computer. Hence *CycleCounter*.

When I was writing *CycleCounter* (CC, for short) I had several goals in mind. I wanted to use it on BASIC as well as machine language programs. It would be nice to see each instruction as it executes, complete with register values and a running total of cycles used. I wanted to be able to stop the program and change the values of the registers. The version of CC presented here incorporates all these features.

*CycleCounter* is located at $C000. It is a little over 3K of ML. Unless you are timing a BASIC program (see the section on timing your BASIC program), you will need a machine language monitor (MLM) to use it. I have tested CC with two monitors, and have had no problems. I have also timed many BASIC programs, and have encountered no problems (however, see the section on the limitations of CC, below).

The easiest way to illustrate the use of CC is to go through a quick example. Let's say your friends at the local CBM users' group swear that the fastest machine language code to move 256 bytes of memory from one location to another takes 3000 cycles. You know you can do it faster, and you want to check your code's speed.

First, load and run your MLM, then load your new code. Finally, load in CC. Using the 'execute code' command from your MLM (usually a 'J' or 'G'), you start up CC. After you give CC some information, it begins to execute your code. When your program has finished, CC prints out a total of all the cycles used. It's as simple as that. And because CC returns control the MLM, you can continue to modify your program.

To use CC from your monitor, simply use the "Execute" or "Jump to" command. A normal *CycleCounter* starts at $C000. Unless you relocate it, you should jump there to begin normal execution. Note that CC uses the BRK command to stop itself and return to your MLM.

When you run *CycleCounter*, it will ask you for information about your program and options. These options and inputs are explained below.

**The start address**

The first thing CC will do is ask you for the address at which it should begin counting cycles (usually, this will be the start of your program). Enter the address as a four digit hex number, e.g. $0800. CC has preprinted the '$' for you.

If you wish to abort CC at this time, simply hit Return.

**The end address**

There are two options for the End Address request. If you wish CC to stop counting and return control to the monitor at a certain address, enter that address. For example, entering $0820 at the prompt will allow CC to count cycles until the PC (the Program Counter) is $0820. At this point, CC will execute a 'BRK' instruction, returning control to the monitor.

The second option for the End Address input is to enter a '**b**'. This tells CC to count cycles until a 'BRK' instruction is encountered. This is useful if you are writing code while in the MLM, and have forgotten the end address of your code.

In either case, an end message giving the total number of cycles used will be printed. As with the Start Address, you can abort CC at this time by simply hitting the Return key instead of choosing an option.

**Executing instructions**

The next question CC will ask you is if you want your code to be executed or not. Enter a '**y**' or '**n**' at the prompt. If CC does not execute your code, it will look at each op-code and add in the lowest possible cycle time to the total number of cycles used. Your instructions will not be 'run' - they are only used to look up cycle times.

When you ask CC to execute your program, however, each instruction of your program is actually 'run', just as if you had executed it instead of CC. The total number of cycles used can differ a great deal from the total you get if you do not choose to execute the code.

The execute mode is much more accurate because this is the only way CC can keep track of register values and other changes that affect your program's cycle time. For instance, with branch instructions, such as BEQ, BNE, etc., the only way to know how many cycles are actually used is to know if the branch is taken. Similarly, for modes such as indexed addressing, CC must be able to calculate the effective address by using the values in the registers. If you did not elect to have CC execute your program, the register values will never change.

Finally, as with the previous two input requests, you may abort CC by simply hitting the Return key at the prompt.

**Print trace**

This prompt allows you to choose whether or not you will see the output that CC can display. If you want to see the output display, enter a '**y**'; otherwise, enter '**n**'. Once again, simply hitting the Return key will abort CC.

The output that CC can produce will look like the following:

```
PC      MNEMONIC  MODE   A  X  Y  ST  SP  CYCLES
```

where

**PC** - is the current Program Counter

**MNEMONIC** - is the 3 letter code for the instruction

**MODE** - is the addressing mode that the instruction uses; e.g. ($FB),Y represents the indirect indexed mode.

**A, X, Y, ST, SP** - are the current values of the accumulator, X and Y registers, status register, and stack pointer, respectively. Values shown are in hex and represent the contents *after* the instruction at the PC has been executed.

**CYCLES** - is the number of cycles used since CC began, including those for the current instruction. Note that the cycle count is a *decimal* value.

For example, we might have the following:

```
$0800 LDA #$00      00 01 02 32 FE 000002
```

which indicates that we have just executed the instruction at $0800, and the value of all registers and cycles are shown as they are after the instruction has executed. If you did not choose the 'execute instructions' option, the values of the registers will not change. Note that only six digit places are given to the current cycle count. This is due to the narrow screen. The final message will print out cycles used with nine digits. Unless your program is very large, it is unlikely that these values will wrap around.

### Setting initial register values

The last input CC needs is the values in the registers when it begins your program. To change a register value, move the cursor under the register label, and type the new value. When you hit Return, the values are recorded, and CC begins. If you simply hit Return when the register display is presented, CC will *not* abort; rather, it will use the current register values.

### The pause mode

While CC is running, you may wish to stop it for a moment. To enter the pause mode, press the F1 key. This will display a '>' prompt so you may enter commands. All commands must be followed by a Return. Any command not recognized will be treated as if you simply hit Return. The following is an explanation of the pause mode commands:

**1. Single step:** hitting Return will cause CC to go on to the next instruction. If CC is *executing* your code, the instruction will be performed. CC will return to pause mode after each instruction.

**2. Continue:** to leave pause mode, enter a '**c**'. This allows CC to run at full speed again.

**3. Toggle trace:** to change the setting of the 'print trace' option, enter a '**t**' at the prompt.

**4. Modify registers:** to display and/or modify current register values, enter an '**r**' at the prompt. You will see:

```
A  X  Y  ST SP
00 01 02 36 F8
```

Change the registers as explained above or Return to accept current values. Of course, you cannot alter the value of the stack pointer (SP).

**5. Wait for address:** Even though execution is hundreds of times slower using CC, the addresses and instructions can scroll by very quickly. Moreover, it can be very tedious to sit through several minutes of listings to observe a certain piece of code in action. The 'Wait' command was designed to avoid these situations.

To use the wait command, enter '**w 0123**' at the pause prompt. The 'w' specifies the Wait command. The '0123' is any four digit hexadecimal address. When CC reaches that address, it will enter the pause mode.

You can combine this command with the 'Toggle Trace' command to skip very long sections of print-out. For instance, if your 'wait' address is deep into the code - that is, it will take a while to reach it - you may want to turn the trace off. This would enable CC to execute much faster (output to the screen slows execution down tremendously), but still allow you to view your selected areas of code.

**6. Quit:** enter a 'q' and CC will issue a BRK command. Control then returns to the calling program.

### Relocating *CycleCounter*

Any machine language utility should be relocatable (since you may want to test your own code at $C000). All the monitors that I have used have a relocation command for code. The basic principle behind them is to modify all absolute address instructions (JMP $C000, LDA $C001, etc.) whose operands are within the range of the old code. Thus, an absolute address instruction that used a location within $C000-$CD00 would be changed (if we were relocating CC). It would not modify any other absolute address instructions, such as those that access the VIC chip at $D000. Consult your monitor manual for the command usage or simply assemble the source with your desired start address.

If you're working with the object code, you will have to do a little more work than just move CC around in memory, however. In order for CC to be relocatable, it has to use pointers to various data areas. These pointers contain absolute addresses that point to locations within the CC code. Whenever you relocate CC, you must modify these pointers as well (there are about 30 or so). This is so tedious that I have included a short routine at the end of CC to do it for you. To relocate CC to a new address, follow these instructions (begin with CC loaded into location $C000):

1) Put the low byte of the new address into location $C00B. Put the high byte of the new address into location $C00C.

2) Use the monitor to execute the code starting at $C006.

3) When control returns from the subroutine, Move (using the memory transfer utility of your monitor) locations $C000-$CC0C to the new location.

4) Using the relocation command of your monitor, relocate all code starting at offset $0452 from your new address. That is, if your new start address for CC is $1000, you will want to relocate all code from $1452 to $1C0C.

5) You will have to modify the first three instructions by hand. If you look at $C000 with the disassembly tool of your monitor (with CC loaded), you will see three successive JMP instructions. You must change the absolute addresses of these instructions to refer to the new addresses of the relocated code. For example, if your relocated CC started at $1000, you will have to change the instruction at $1000 from 'JMP $C452' to 'JMP $1452' (you will have to change the other two as well).

6) Save the new version of CC.

### Using *CycleCounter* with BASIC programs

One of the more interesting features of CC is the ability to count the number of cycles your BASIC program uses. In order to exploit this ability, you should use the following procedure:

First, load in CC using a secondary address of one:

```
load "cyclecounter.02",8,1
```

(and remember to enter 'new' to reset the pointers). Use the version located at $C000, leaving the normal BASIC RAM free.

Now SYS 49155. When the BASIC prompt returns, type RUN. You will enter CC.

The request for the starting address will be skipped. When CC asks for an ending address, *you must enter $A480*. This is the address of the top of BASIC's main loop. When the PC is equal to this, CC will know that your BASIC program has finished, and it will return control to BASIC. Answer the remaining questions to suit your preference.

Note that if you do not have CC *execute* the code, *there is no guarantee that you will encounter the address of $A480*. In addition, it is not advisable to print a trace out of the whole execution. The operating system appears to encounter problems with the additional text on the screen.

### Limitations of *CycleCounter*

When designing CC, I wanted to keep it as transparent as possible. The optimal utility would not interfere at all with the user's program. For the most part, CC follows this guideline. There are times, however, when CC must utilize the operating system; possibly jeopardizing a user's program. For example, CC makes calls to the system routine CHROUT. This routine is a general routine for outputting characters to a device. CC assumes that the screen is always available, and so makes no attempt to check if your program changed this. CC also makes calls to CHRIN, the general character input routine. The only way to maintain complete transparency, while still utilizing these system routines, is a method known as a 'virtual system'. Essentially, what would occur is that CC would keep a copy of all important system variables for itself, and load them into place every time it used the system, saving the user's variables. This is expensive and time consuming - but possible in theory. I opted, however, to keep CC as small and as fast as possible.

Conflict with operating system requests is not the only problem that may occur. Recall that the speed of CC's execution mode (where your code is actually being executed) is several hundred or thousands times slower than normal execution speed. Thus, if your code is interacting with any time-sensitive devices (disk drive, timers, etc.), you should not use the execution option. Instead, you may wish to change the address of any memory mapped registers your program accesses, to normal memory addresses. This way you can still check the runtime speed, without actually changing any hardware values. This could prevent a lock-up or similar disaster.

### How *CycleCounter* works

Some of you may be curious about how CC works. It is difficult to learn much in assembler by typing in hex codes. A disassembly helps, but often a programmer's best tricks use the computer in a way that is not immediately obvious. I will discuss some of the techniques that I used in CC that may be of interest. I hope that this will give you new ideas or spark an interest in using them in your own code.

To start with, CC has over 1K of data. Much of the size of CC is accounted for by input/output routines: formatting routines for op-code print-out, etc. There is also a data table for the op-codes. Each op-code has one byte, and the groups of bits within each mean something to CC. For instance, bits zero through two are used to give the basic cycle count for that op-code to execute. To get information about a specific op-code, we use a 'look-up table'; that is, we use the op-code as an index into the table, and look at the bits we need.

To execute one of your program's instructions, CC copies the instruction into its own code area. After this, the user's six registers are reloaded with the values they had after the last instruction from your program. The instruction is then executed by simply letting CC run into it, as if it had always been there. Actually, before CC copies your code into its own instruction area, it puts three NOPs in a row. This works out nicely, since the maximum length of any 6510 instruction is three bytes. If an instruction is less than that, it will be executed, followed by one or two NOPs. Since the NOP does nothing, the status register and memory areas of your program are unaffected. If we did not clear out the instruction area with NOPs, we might execute the data or instructions left over from the last op-code. This is an example of self-modification, as discussed above.

After your program's instruction has been executed, CC takes over again, saving all your registers before continuing.

There are several instructions that CC does not copy into its own program area; JMP absolute is one. To see why, consider what would happen if CC copied a JMP $0800 into its own program area, and then executed that instruction. Control would pass to whatever code is located at $0800 - CC has lost control of the processor!

To avoid this situation, such instructions are simulated by CC. For example, a JSR $1000 command, executed normally, pushes the current value of the PC + 2 onto the stack, and then loads the PC with $1000. Since CC keeps track of the user's PC, it is short work to change it, and then push some data onto the stack. An RTS instruction is likewise simulated: CC pulls the return address from the stack, and changes the copy of your program's PC.

The point is: CC never loses control of the processor - each instruction that would cause this to happen has a special routine to simulate its effect on the processor and programming environment. In fact, even the BRK instruction is simulated.

Earlier, I mentioned the concept of a 'virtual system', where CC would keep all the important system variables in a data area, swapping them back and forth with your program's system variables (system variables include such things as what files are open and where the current text screen is located). *CycleCounter* does this with the registers, but it also does a small amount of swapping of memory as well. It uses some of the free zero page space (addresses $FB-$FF) for its own variables. It would not be a very useful utility if you could not use these zero page locations yourself, so CC keeps two copies: one for you and one for itself.

It works like this: Up until CC actually executes your next program statement (or simulates it), it is using locations $FB-$FE for itself. Right before your program instruction is done, it replaces these memory locations with your copy of $FB-FE. After your instruction is done, it copies your values of $FB-$FE into its data area, and moves its own copy of $FB-$FE back in. Your program never notices that these memory locations are shared by two programs. This is exactly like saving and restoring the registers.

**Using *CycleCounter* as a programming tool**

Now it is time to show how we can use CC to help in the development of faster code.

Despite its great graphic capabilities, the C64 lacks system-supported graphics primitives. There are no commands to draw lines, dots, squares, or anything. Instead, the programmer must manipulate the bits and bytes of the display screen memory. The biggest problem with this approach, at least in BASIC, is the lack of speed. The drawing of images is agonizingly slow.

The *C64 Programmer's Reference Guide* offers a simple algorithm to light a pixel on a hi-res screen (see pages 125-126 of the *PRG*). This is a very important routine, since almost any graphics routine that draws on the screen will have to use it. Since it will be used so often, it should be as fast as we can make it. Our goal will be to develop the fastest pixel-lighting routine possible.

In BASIC, we can reduce the *PRG* code to the two-line subroutine (lines 100 and 110) in the short program below. Note that the variable BA is the base (or start) address of the hi-res screen in memory, and the X and Y variables represent the coordinates of the pixel on the screen. Lines 10 and 20 set and make the call to the pixel plotting routine:

```
10 ba = 8192:x = 50:y = 50
20 gosub 100:end
100 by=ba+(int(y/8)*320)+(8*(int(x/8))+(yand7)
110 pokeby,peek(by)or(2^(7-(xand7))):return
```

Using CC to time this short program, we get the following cycle counts: Timing 1, 85646. Timing 2, 148620

There are two points of interest here. The first is the number of cycles used! As we shall soon see, this is a staggering amount of processing time to spend on such a (seemingly) simple operation. BASIC users trade speed for the usefulness of an interpreted language.

The second interesting point is that there are two different times listed for the one program. In the first timing, I made sure the screen was clear before using CC. I then typed RUN at the top of the screen. In the second timing, I typed RUN on the bottom line of the screen. After both trials finished, BASIC printed 'READY.' to the screen. When the second timing ended, however, BASIC had to scroll the screen upwards before printing the READY message. This accounts for the large difference in timings. Printing to the screen has never seemed so expensive! From now on, I will only discuss timings achieved when the screen has been cleared, and scrolling is not necessary.

Anyone who has programmed in machine language is aware of the increase in speed over equivalent BASIC programs. Listing 3 at the end of this article is an ML program that was developed by following the BASIC subroutine.

Reproducing the exact equations used in the BASIC subroutine would be very tedious. To avoid this, I first reduced as many of the expressions as possible to lowest terms. For instance, the equation

```
(8 * (int (x / 8)))
```

can also be written as

```
(x and 248)
```

provided the value of **x** is between zero and 255.

The equation involves a multiplication by 40. To achieve this, I used the simple technique of adding the multiplicand to itself 40 times.

Once the correct byte has been located, the position of the bit within that byte must be determined. The BASIC method involves exponentiation - specifically, raising two to a small integer power. This can be done very efficiently in machine language by shifting a single bit left within a loop.

In order to allow meaningful comparison with the BASIC program above, we need a few lines of BASIC from which to call the machine language:

```
10 x = 50: y = 50
20 gosub 100:end
100 poke 782,y
110 if x>255 then poke 780,1:poke 781,(x and 255):goto 130
120 poke 780,0:poke 781,x
130 sys32768:return
```

This program uses the fact that when the SYS command is issued from BASIC, it loads the A, X, and Y registers with the values found in addresses 780-782. Then it passes execution to the machine language subroutine. Since our subroutine requires the registers to contain the proper values, we can simply poke those values into memory and let the SYS command load them for us. Of course, this method is not as general as the BASIC version, since we cannot simply change the location of the screen by modifying a variable. However, notice that each machine language subroutine loads a value that is the high byte of the start-of-screen-address (no low byte is needed since the screen must start on the beginning of a page, e.g. $2000). This is performed by the LDA #$20 instruction in each routine. Should the need present itself, you could simply poke a new value into the instruction area. Note, however, that this location is different for each routine.

With the code from Listing 3 located at $8000 (decimal 32768), we can now use CC to time the total number of cycles used. The minimum number of cycles needed to complete this routine, including an extra 6 cycles for a JSR, is 829. The maximum number of cycles this routine will ever need is 998 cycles. Why are there multiple timing values for this routine? Recall that we are using loops and addition in this routine. Some values of the X and Y coordinate will cause the loop at loop2 to be executed seven times, while others will require only one pass through it. Also, the addition we are using in the main loop, loop1, is double precision. Although this loop is executed the same number of times for each call to this routine, different values of the Y coordinate (for example, very large numbers) will cause the INC PNTR+1 instruction to be executed more frequently. Small values of Y can get through the loop without ever using this instruction. Thus, in this subroutine at least, the values of the pixel coordinates can affect the timing - if only by a hundred millionth of a second, or so.

When we time the BASIC program that calls the routine in Listing 3, we find it uses 43686 cycles to complete. This betters the speed of the original BASIC pixel plotter by about 49%.

Computers can perform feats of mathematics far faster than any of us could hope to. Compared to other functions, however, the instructions used to perform mathematics can be very slow. In any situation where you are trying to improve the speed of your code, you should try to exploit as much knowledge about the problem and computer as possible. For instance, the following lines of machine language perform the math function of multiplication by four:

```
clc       ;prepare for add
lda value ;we'll multiply value by 4
adc value ;acc = value * 2
adc value ;acc = value*3
adc value ;acc = value * 4
sta value ;store result
```

There are two assumptions to bear in mind for this short segment of code. The first is that *value* is not located on zero page. If it were, fewer cycles would be needed to perform this code. The second is that *value* * 4 will not exceed 255, since we make no provision for handling a carry.

The total time taken by this code segment is 22 cycles. Now look at the following lines of code:

```
asl value ;shift left = * 2
asl value ;shift left = * 4
```

Like the previous example, this segment of code multiplies a number by four (the same assumptions apply). This time, however, instead of using the standard addition instruction, ADC, we used the ASL (shift left one bit) instruction. We have exploited the fact that moving a binary number two places to the left is the same as multiplying it by four. The timing for this segment of code is now only 12 cycles. This comes close to halving the number of cycles (in addition to reducing the number of instructions by a factor of three) needed by a more straightforward approach. Simple observations like this can often lead to faster and more efficient code.

Listing 4 contains the assembly code for another subroutine that sets a pixel on a hi-res screen. You will notice that it bears little resemblance to Listing 3. This is because it uses information about C64 hi-res screens that Listing 3 did not. This extra information is held in a table at the end of FASTPX. The data are based on several observations.

First, in the previous pixel-lighting algorithm, we spent a lot of time multiplying the value of Y to obtain the correct address. This is necessary because of the non-linear method by which screen memory is displayed (you may wish to refer to pages 122-127 in the *PRG* for a more complete description of the standard hi-res mode). The second observation is that, instead of multiplying, part of the address can be obtained by using the Y value as an index into a table of offsets. This table is in two parts: the low bytes of the offsets and the high bytes of the offsets. By adding in the high and low byte of the offset to a base address, FASTPX quickly calculates the address of the leftmost pixel for the row.

For instance, to calculate the leftmost pixel in row 25, first load the A register with the value at 'scrlo+25'. Add this value to the low byte of the hi-res screen base address. Next load A with 'scrhi+25', and add it to the high byte of the base address. This resulting address is the leftmost byte (where X values range from 0 - 7) at row 25. It is then a simple matter to add in the value for the X component of the pixel.

Note that this scheme uses the upper left corner of the screen as the origin (location 0,0). If you want to change this, simply reorder the data in the table (you can achieve some interesting effects by scrambling the data in the table, making the lighting of pixels unpredictable). Notice, as well, that I forced each half of this data table to be located at the start of a page ('scrlo' is at $8100 and 'scrhi' at $8200). While this is a waste of memory, it was done to demonstrate a point. If the first half (scrlo) of the table was simply placed into memory right after the code, there is a very good chance that part of it would lie on the same page as the code, and the rest on the next page. Since the combined size of scrhi and scrlo is more than 400 bytes, a similar page split might happen to scrhi. The indexed addressing mode is used to access the table, which means that cycles are saved if the index does not cross a page when memory is accessed. Forcing alignment on a page was the only way to insure the minimum cycle time would be used for access to this data.

There are other differences between this code and the code in Listing 3. Instead of using a loop to shift a single bit over, the correct bit to set is calculated from a look-up table. The lower three bits of the X register are used as an index into this table. Since there are eight pixels to one byte on the hi-res screen, the bytes in the table each have one of the possible eight bits set. Thus we can simply load this value in and use it.

The number of cycles used by this subroutine is only 75, including six cycles for a JSR instruction to get us into the code. Note that, unlike the previous routine, this 75 cycles does not change with the value of the coordinates. We are guaranteed 75 cycles every time we call it.

The cycle count for this listing is quite a bit less than the number used by Listing 3. Again, we standardize the cycle count by calling the new routine from BASIC. Using the same program as we did for Listing 3, but replacing the code at $8000 with the code in Listing 3, we get 42418 cycles, an improvement of about 50% over the original BASIC program.

The final version of a pixel plotter is shown in Listing 5. This represents a compromise between the versions in Listings 3 and 4. It does not perform a straight calculation as in Listing 3, nor does it have a huge table of data as in Listing 4.

This version exploits two more facts about the way the graphic screen is organized. The first fact is easy to pick up if you study the data table used by Listing 4: the low-order bytes repeat themselves every 32 bytes. Thus, all we need is a small table to hold these 32 bytes, and some new way to calculate the high order byte for the offset. As it turns out, this is the second fact of graphic screen layout that we can exploit. Part of the high order byte can be calculated by simply using the upper five bits of the value passed to us in the Y register! By manipulating these bits, we can obtain the high order value for the leftmost byte offset.

This 'manipulation' may seem very obscure at first. Examine the 'scrhi' table of Listing 4 until you notice a trend. There are two observations we can make from these bytes. First, each byte occurs eight times in a row. Second, the value of one is added to each number after it has repeated - except after 32 bytes have gone by; then two is added. To see what I mean, notice that 'scrhi' starts off with eight zeroes. Then we add one, and we have eight ones. Add one again, and we have eight twos. Add one more to the byte value of the last row, and we have eight threes. That makes 32 bytes so far. Look at the next line - it contains eight fives. So, every 32 bytes, we add two to the current byte value in the table.

While this routine is quite a bit smaller than Listing 4, it is slightly slower. The number of cycles needed to execute is 114, including the JSR. The number of cycles is constant for any pair of coordinates.

Again, for standardization, we will use the BASIC program we used with Listing 3. In Listing 5, however, the usage of registers A and X is reversed, so the references to 780 and 781 in the BASIC code must be interchanged. Using CC to time this program, we find that it takes 42456 cycles to complete. This is still an improvement of about 50% over the original BASIC.

These examples are not meant to be the final word on pixel plotters. In fact, I would enjoy seeing routines that could perform this function faster than the ones shown. The real purpose of these examples is to show how CC was used as a tool. I was able to quickly count cycles, modify the routines, and count again. These routines are, for the most part, too fast to be timed by other means, and the timing from BASIC would be nearly impossible to count by hand. Cycle counting allowed me a basis for comparison that I did not have before.

### Some final notes on *CycleCounter*

The major concept now is one involving the conversion of cycles into a time value. While *CycleCounter* will give you a precise number of cycles needed to execute your program, it cannot tell you precisely how long (in terms of seconds and microseconds) it will take your program to run. There are several reasons for this. An obvious one, especially for programmers, is the IRQ that occurs every 1/60 of a second. The C64 operating system causes one of the hardware timers to interrupt normal program execution, saving the current state of your program, and then performing necessary chores like reading the keyboard, updating clocks, etc. When the operating system has finished the interrupt, it restores your program to its previous state, and allows it to run again. Your code will be slowed by this constant, but transparent, interruption.

Another reason an accurate time value cannot be given is due to a quirk in the relationship between the 6510 and the VIC II video chip. There are times - when sprites are on the screen, for example - when the video chip must use the system data bus more than normal to maintain the display screen. When this occurs, the chip will 'steal' the bus from the 6510 processor for a short time. This causes a slight loss of processing speed.

For these and other reasons, CC cannot give you a precise execution speed in, say, seconds. What does CC give you then? The first way you can use CC is as a speed analysis tool. Regardless of any other interruptions, the code you write will be executed in a time that is directly proportional to the number of cycles that code uses. Put another way: more cycles needed, more time needed. You could thus say that, in most cases, the code with the fewer cycles will run faster.

Secondly, although CC cannot give a accurate timing in seconds, you can get a close estimate by converting cycles to seconds. The actual clock speed of the C64 is 1,022,730 cycles/second using NTSC, and 985,250 cycles/second for those using PAL. So, if your program took 100 cycles, you could divide 100 by the number of cycles per second to get an execution value in seconds. To get an even more accurate value, you could count the number of cycles the operating system interrupt code uses and add that in to your total cycle time. Of course, this varies, depending on such things as key presses and RS-232 activity.

If you are interested, the C64's hardware timers (in the 6526 CIAs) have the capability to count the cycles used as time goes by. While this may seem an appealing way to compare programs, it introduces some new problems. For instance, the only way to be sure that every program you test is getting the same treatment is to reset before each test. Furthermore, since you do not have control of things like the system clock or IRQ timing, there is no way to keep out interruptions that could slow down your execution. There are ways around this, of course, but the main point I wish to make is that one must beware of hidden variables: When you are dealing in microscopic time values like cycles, it is very difficult to compare program execution times by actually running them. Nevertheless, this type of utility could have many uses - but that is easily the subject of another article.

*CycleCounter* avoids some of the timing problems by abstracting each instruction to its best possible case, and allowing you to compare programs based on that. If the computer had no other chores to do - no IRQs, no bus stealing - then the execution value in seconds you can calculate should be an accurate one.

In closing, I hope that *CycleCounter* becomes a useful addition to your programming toolbox. The need for speed is never ending, especially as the smaller eight-bit computers are forced to compete with the faster 16-bit and 32-bit processors. It is up to all of us to make every cycle count.

**Listing 1. This subroutine uses 3342 cycles to complete.**

```
fill1    stx $fb       ;save low byte of block in zero page
         sty $fc       ;save high byte of block in zero page
         lda #$00      ;Acc. holds value to store in block
         ldy #$00      ;set index register to zero
         sta ($fb),y   ;do first byte in block
;
loop1    iny           ;move index pointer up one
         sta ($fb),y   ;stuff value of Acc. into memory
         cpy #$ff      ;when Y=FF we are done
         bne loop1     ;not done yet...
         rts           ;now we're done, return to caller
```

**Listing 2. This subroutine uses 2583 cycles to complete.**

```
fill2    stx loop+1    ;save low byte into STA instruction
         sty loop+2    ;save high byte into STA instruction
         lda #$00      ;Acc. holds value to store in block
         tay           ;set index register to zero
;
loop2    sta $0000,y   ;save value of Acc. into memory
         iny           ;move index pointer up one
         bne loop2     ;keep going till INY sets zero flag
         rts           ;now we're done, return to caller
```

**Listing 3. Simple ML Pixel Plotter**

```
;           BASPIX
;
;  This ML subroutine is a very literal translation
;of the pixel plotting routine found on pages 123-124
;of the CBM Programmers Reference Guide.
;Very little optimization has been done.
;
; min cycles to execute: about 829
; max cycles to execute: about 998
;
; includes 6 cycles for a JSR to get us here
;
;============================
;=    register usage on entry    =
;============================
;
; a = 0 if x<255, 1 if x>255
; x = 0 - 255
; y = 0 - 199
;
pntr = $fb         ;a zero page pointer
;
```

```
        * = $8000
;
baspix sta pntr+1   ;add in a right away
       txa          ;put x into a
       pha          ;save x for later
       tya          ;save y for later use
       pha          ;push y on stack
;
; here, we will calculate the
;   (8 * (int (x / 8)))
; note that this is the same as
;   (x and 248)
       txa          ;now a = x
       and #$f8     ;now a = x and 248
       sta pntr     ;save it
;
; now we are going to perform the
;calculation of
;     int (y / 8) * 320
; note we can do this as
;     (y and 248) * 40
nxtlp2 tya          ;move y into a
       and #$f8     ;mask off lower bits
       tay          ;use this value to add
       ldx #40      ;prepare to add in 40 times
loop1  tya          ;move value to add in 39 times into acc.
       clc          ;ready to add now...
       adc pntr     ;adding in y 40 times is same as y * 40
       sta pntr     ;hold onto the new value!
       bcc nxtlp1   ;if no carry, skip next instruction
       inc pntr+1   ;add 1 to hi byte of pointer
nxtlp1 dex          ;are we done yet?
       bne loop1    ;no! keep going!
;
       pla          ;get back value of y from stack
       and #$07     ;this gives us y and 7
       clc          ;add this in, as well!
       adc pntr     ;add it
       sta pntr     ;save it
       bcc nxtlp3   ;skip it, if no carry...
       inc pntr+1   ;add 1 to hi byte
;
; now we add in the base address of
;our hi-res screen. we only need the
;high byte, since it must start on
;on the beginning of a page
nxtlp3 clc          ;prep for add
       lda #$20     ;screen located at $2000
       adc pntr+1   ;add it in
       sta pntr+1   ;and save it
;
; now, pntr and pntr+1 point to
;the byte we want in screen memory
;here, we calculate which bit we
;are going to turn on.
       pla          ;get x from stack
       and #$07     ;gives us x and 7
       sta bitlit   ;save for our subtraction
       lda #$07     ;we'll subtract bit from 7
       sec          ;prep for subtraction
       sbc bitlit   ;gives us a = 7 - (x and 7)
       tay          ;we'll use this value as a counter
;
; we now use (x and 7) as a counter
;for shifting a single bit to the
;left.
       sec          ;first time thru, carry gets pushed into acc.
       lda #$00     ;start out with 0 in acc.
;
loop2  rol a        ;shift single bit to left
       dey          ;are we done yet?
       bpl loop2    ;not yet, keep going!
; finally, we can access our byte
;and turn our bit on
       ldy #$00     ;make our index = 0
       ora (pntr),y ;turn on our bit
       sta (pntr),y ;and save the change
       rts          ;all done..
;
bitlit .byte 0      ;a variable
       .end
```

## Listing 4. Fast ML Pixel Plotter

```
;           FASTPX
;
;  FASTPX is a subroutine that is
; designed to light up a dot on the
; hi-res screen in about 75 cycles,
; including six cycles for a jsr to
; get us here.
;  It should be called in the
; following manner:
;
; y  =  (0-199)  y axis range
; x  =  (0-255)  x axis range
; a  =  (0 or 1) if x > 255
;
pntr = $fb      ;zero page pointer
;
          * = $8000
;
; first, calculate the offset for
; the row
fastpx sta pntr+1  ;store upper byte of x
       lda scrlo,y ;get lo byte of offset
       sta pntr    ;set up lo byte of pointer
       clc         ;prepare for addition
       lda scrhi,y ;get hi byte of offset
       adc pntr+1  ;add to hi x
       sta pntr+1  ;set up hi byte of offset
;
; now we can calculate what bit we
; will light up at our address..
       txa         ;move in lo byte of x
       and #$07    ;get the bit to light up
       tay         ;save it in y
;
; here we calculate the address
; of byte to modify, by adding in
; x coordinate and screen base
       txa         ;get lo byte of x
       and #$f8    ;make it a power of 8
       adc pntr    ;add x to offset
       sta pntr    ;update offset
;
; now add the hi byte and screen base
       lda #$20     ;screen base at $2000
       adc pntr+1   ;add to offset
       sta pntr+1   ;update offset
;
; we now have the address to modify
; let's light up a bit in that byte!
       lda table,y  ;pick up the bit position to lite
       ldy #$00     ;set y index to 0
       ora (pntr),y ;lite up that bit
       sta (pntr),y ;and make change permanent
       rts          ;go home...
;
table  .byte $80,$40,$20,$10,$08,$04,$02,$01
;
```

```
        * = $8100
;
;    this data file contains
; offsets to add to a screen base
; address. these offsets will give
; you the address of the leftmost
; byte, in the y position. use the
; y value for an index
;
scrlo .byte $00,$01,$02,$03,$04,$05,$06,$07
      .byte $40,$41,$42,$43,$44,$45,$46,$47
      .byte $80,$81,$82,$83,$84,$85,$86,$87
      .byte $c0,$c1,$c2,$c3,$c4,$c5,$c6,$c7
;
      .byte $00,$01,$02,$03,$04,$05,$06,$07
      .byte $40,$41,$42,$43,$44,$45,$46,$47
      .byte $80,$81,$82,$83,$84,$85,$86,$87
      .byte $c0,$c1,$c2,$c3,$c4,$c5,$c6,$c7
;
      .byte $00,$01,$02,$03,$04,$05,$06,$07
      .byte $40,$41,$42,$43,$44,$45,$46,$47
      .byte $80,$81,$82,$83,$84,$85,$86,$87
      .byte $c0,$c1,$c2,$c3,$c4,$c5,$c6,$c7
;
      .byte $00,$01,$02,$03,$04,$05,$06,$07
      .byte $40,$41,$42,$43,$44,$45,$46,$47
      .byte $80,$81,$82,$83,$84,$85,$86,$87
      .byte $c0,$c1,$c2,$c3,$c4,$c5,$c6,$c7
;
      .byte $00,$01,$02,$03,$04,$05,$06,$07
      .byte $40,$41,$42,$43,$44,$45,$46,$47
      .byte $80,$81,$82,$83,$84,$85,$86,$87
      .byte $c0,$c1,$c2,$c3,$c4,$c5,$c6,$c7
;
      .byte $00,$01,$02,$03,$04,$05,$06,$07
      .byte $40,$41,$42,$43,$44,$45,$46,$47
      .byte $80,$81,$82,$83,$84,$85,$86,$87
      .byte $c0,$c1,$c2,$c3,$c4,$c5,$c6,$c7
;
      .byte $00,$01,$02,$03,$04,$05,$06,$07
      .byte $40,$41,$42,$43,$44,$45,$46,$47
;
       * = $8200
;
scrhi .byte 0,0,0,0,0,0,0,0
      .byte 1,1,1,1,1,1,1,1
      .byte 2,2,2,2,2,2,2,2
      .byte 3,3,3,3,3,3,3,3
;
      .byte 5,5,5,5,5,5,5,5
      .byte 6,6,6,6,6,6,6,6
      .byte 7,7,7,7,7,7,7,7
      .byte 8,8,8,8,8,8,8,8
;
      .byte $0a,$0a,$0a,$0a,$0a,$0a,$0a,$0a
      .byte $0b,$0b,$0b,$0b,$0b,$0b,$0b,$0b
      .byte $0c,$0c,$0c,$0c,$0c,$0c,$0c,$0c
      .byte $0d,$0d,$0d,$0d,$0d,$0d,$0d,$0d
;
      .byte $0f,$0f,$0f,$0f,$0f,$0f,$0f,$0f
      .byte $10,$10,$10,$10,$10,$10,$10,$10
      .byte $11,$11,$11,$11,$11,$11,$11,$11
      .byte $12,$12,$12,$12,$12,$12,$12,$12
;
      .byte $14,$14,$14,$14,$14,$14,$14,$14
      .byte $15,$15,$15,$15,$15,$15,$15,$15
      .byte $16,$16,$16,$16,$16,$16,$16,$16
      .byte $17,$17,$17,$17,$17,$17,$17,$17
;
      .byte $19,$19,$19,$19,$19,$19,$19,$19
      .byte $1a,$1a,$1a,$1a,$1a,$1a,$1a,$1a
      .byte $1b,$1b,$1b,$1b,$1b,$1b,$1b,$1b
      .byte $1c,$1c,$1c,$1c,$1c,$1c,$1c,$1c
;
      .byte $1e,$1e,$1e,$1e,$1e,$1e,$1e,$1e
      .byte $1f,$1f,$1f,$1f,$1f,$1f,$1f,$1f
```

## Listing 5. Optimized ML Pixel Plotter

```
;          QUIKPX
;
;   This subroutine is designed to
; light a pixel on a hi-res screen
; in 114 cycles. This includes six
; cycles for a JSR to get us here.
; Values must be sent as follows:
;
; a = low x  (0   - 255)
; x = high x (0 if a<255, else 1)
; y = y      (0   - 199)
;
pntr = $fb   ; low core pointer
;
       *=$8000
;
;  first, we save stuff to use later
;
quikpx pha          ;save low part of x
       tya          ;move row # into a
       pha          ;save row
;
; now we use fact that low byte of
; row offset repeats every 32 bytes
;
       and #$1f     ;get number range of 0-31
       tay          ;now use as an index
       lda lookup,y ;get lo byte of left column of row
       sta pntr     ;set it up in lowcore
;
;  here, we calculate the high byte
; of the row offset from screen base
; by munging on the bits in y
;
       pla          ;get row
       clc          ;make it ok to shift bits
       and #$f8     ;dont move anything into carry
       ror a        ;divide by 2
       ror a        ; ...  by 4
       ror a        ; ...  by 8
       sta pntr+1   ;upper byte of screen offset for y value
;
       and #$fc     ;to prevent shifts into the carry
       ror a        ;divide by 16
       ror a        ;divide by 32
       adc pntr+1   ;add to upper byte
       sta pntr+1   ;make it official
;
; now, we save the bit position we
; will light up when after we calculate
; the byte address.
;
       pla          ;get low byte of x
       tay          ;hold it temporarily
       and #$07     ;these will be the bits to light in byte
       pha          ;save them
;
; here, add in the value of the x
; coordinate.
;
       tya          ;get back low byte of x
       and #$f8     ;make it a power of 8 (max of 224)
       adc pntr     ;carry still clear, add it
       sta pntr     ;make change official
;
; now we add in the base of our
; hi-res screen
;
       txa          ;get hi byte of x value
       adc #$20     ;start of hi-res screen--hi byte
       adc pntr+1   ;add to hi byte of offset
       sta pntr+1   ;make it official
```

```
;
; get back our pixel position, and
; use it to look up value.
;
        pla          ;get bit to light in byte
        tax          ;use as index into table of bytes
        lda litbit,x ;get value of byte to use
;
; and finally, we are ready to light
; up that pixel!
;
        ldy #$00     ;set index = 0
        ora (pntr),y ;create a new byte
        sta (pntr),y ;store it !
;
        rts          ;jump home
;
litbit .byte $80,$40,$20,$10,$08,$04,$02,$01
;
lookup .byte $00,$01,$02,$03,$04,$05,$06,$07
       .byte $40,$41,$42,$43,$44,$45,$46,$47
       .byte $80,$81,$82,$83,$84,$85,$86,$87
       .byte $c0,$c1,$c2,$c3,$c4,$c5,$c6,$c7
       .end
```

## Listing 6. Generator for CycleCounter at $C000

```
AM 100 rem  generator for "cc.c000"
FJ 110 nd$="cc.c000": rem  name of program
CG 120 nd=3084: sa=49152: ch=331732
KO 130 for i=1 to nd: read x
EC 140 ch=ch-x: next
ID 150 if ch<>0 then print"data error": stop
FF 160 print"data ok, now creating file": print
CM 170 restore
HH 180 open 8,8,1,"0:"+f$
CF 190 print#8,chr$(sa/256)chr$(sa-int(sa/256));
AD 200 for i=1 to nd: read x
CF 210 print#8,chr$(x);: next
EM 220 close 8
CO 230 print"prg file '";f$;"' created..."
CH 240 print"this generator no longer needed."
OG 250 :
IL 1000 data  76,  82, 196,  76, 206, 202,  76, 205, 203,   0, 192,   0
FF 1010 data   0,   0,   0,   0,   0,   0,   0,   0,   1,   2,   0,   0
JK 1020 data   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  79,  46
IP 1030 data   0,   0,   0,  19,  21,   0,  75,   2,  26,   0,   0,  12
OP 1040 data  14,   0,  34, 101,   0,   0,   0,  60,  62,   0,  74, 220
GI 1050 data   0,   0,   0, 212,  87,   0,  14,  46,   0,   0,  19,  19
IB 1060 data  21,   0,  76,   2,  26,   0,  12,  12,  14,   0,  34, 101
EH 1070 data   0,   0,   0,  60,  62,   0,  74, 220,   0,   0,   0, 212
AF 1080 data  87,   0,  78,  46,   0,   0,   0,  19,  21,   0,  75,   2
LC 1090 data  26,   0,  11,  12,  14,   0,  34, 229,   0,   0,   0,  60
IH 1100 data  62,   0,  74, 220,   0,   0,   0, 212,  87,   0,  78,  46
EA 1110 data   0,   0,   0,  19,  21,   0,  76,   2,  26,   0,  53,  12
BG 1120 data  14,   0,  34, 229,   0,   0,   0,  60,  62,   0,  74, 220
CC 1130 data   0,   0,   0, 212,  87,   0,   0,  46,   0,   0,  19,  19
OH 1140 data  19,   0,  74,   0,  74,   0,  12,  12,  12,   0,  34, 102
FC 1150 data   0,   0,  60,  60,  68,   0,  74,  93,  74,   0,   0,  85
KN 1160 data   0,   0,   2,  46,   2,   0,  19,  19,  19,   0,  74,   2
DN 1170 data  74,   0,  12,  12,  12,   0,  34, 229,   0,   0,  60,  60
BI 1180 data  68,   0,  74, 220,  74,   0, 212, 212, 220,   0,   2,  46
KE 1190 data   0,   0,  19,  19,  21,   0,  74,   2,  74,   0,  12,  12
BL 1200 data  14,   0,  34, 229,   0,   0,   0,  60,  62,   0,  74, 220
GH 1210 data   0,   0,   0, 212,  87,   0,   2,  46,   0,   0,  19,  19
NM 1220 data  21,   0,  74,   2,  74,   0,  12,  12,  14,   0,  34, 229
EB 1230 data   0,   0,   0,  60,  62,   0,  74, 220,   0,   0,   0, 212
OB 1240 data  87,   0,   1,   2,   1,   0,   1,   1,   2,   1,   1,   0
II 1250 data   2,   2,   1, 176, 144, 112,  80, 208, 240,  16,  48,  32
DN 1260 data  96,  76, 108,  64,   0,  36,  32,  32,  32,  32,  32,  32
CK 1270 data  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32
MK 1280 data  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32
PB 1290 data  32,  32,  32,  32,  32,  32,  32,  32,  13,   0,   0,   0
JN 1300 data   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  32
FD 1310 data  32,  32,  32,  32,  32,  32,  32,  32,  32,   0,   0,   0
LG 1320 data  48,  49,  50,  51,  52,  53,  54,  55,  56,  57,  65,  66
CC 1330 data  67,  68,  69,  70,   0,   0,   0,   0,   0,   0,  95, 197
BB 1340 data  95, 197,  95, 197,  95, 197,  95, 197,  95, 197,  95, 197
CP 1350 data  95, 197, 236, 196,  19, 197,  46, 197,  57, 197, 156, 197
FD 1360 data 111, 197,  23, 192,  61, 193, 105, 193,  40, 196,  31, 192
BH 1370 data  29, 192, 231, 202,  61, 193,  63, 193,  13, 192, 157, 196
EH 1380 data  79, 193,  31, 195, 255, 255, 109, 195, 147, 195, 170, 195
NN 1390 data 193, 195, 228, 195,  14, 196, 224, 193,  46, 196, 253, 195
DB 1400 data  65,  68,  67,  65,  78,  68,  65,  83,  76,  66,  67,  67
DA 1410 data  66,  67,  83,  66,  69,  81,  66,  73,  84,  66,  77,  73
OA 1420 data  66,  78,  69,  66,  80,  76,  66,  82,  75,  66,  86,  67
OC 1430 data  66,  86,  83,  67,  76,  67,  67,  76,  68,  67,  76,  73
GD 1440 data  67,  76,  86,  67,  77,  80,  67,  80,  88,  67,  80,  89
AI 1450 data  68,  69,  67,  68,  69,  88,  68,  69,  89,  69,  79,  82
NF 1460 data  73,  78,  67,  73,  78,  88,  73,  78,  89,  74,  77,  80
HF 1470 data  74,  83,  82,  76,  68,  65,  76,  68,  88,  76,  68,  89
GD 1480 data  76,  83,  82,  78,  79,  80,  79,  82,  65,  80,  72,  65
HB 1490 data  80,  72,  80,  80,  76,  65,  80,  76,  80,  82,  79,  76
GE 1500 data  82,  79,  82,  82,  84,  73,  82,  84,  83,  83,  66,  67
OG 1510 data  83,  69,  67,  83,  69,  68,  83,  69,  73,  83,  84,  65
OI 1520 data  83,  84,  88,  83,  84,  89,  84,  65,  88,  84,  65,  89
PI 1530 data  84,  83,  88,  84,  88,  65,  84,  88,  83,  84,  89,  65
PM 1540 data  10,  34,  34,   2,  36,  34,   2,  34,   2,   9,  34,  34
OM 1550 data   2,  13,  34,  34,   2,  28,   1,   6,   1,  39,  38,   1
PI 1560 data  39,   5,   1,  39,   7,   1,   1,  39,  44,   1,   1,  39
HN 1570 data  41,  23,  23,  32,  35,  23,  32,  27,  23,  32,  11,  23
NF 1580 data  23,  32,  15,  23,  23,  32,  42,   0,   0,  40,  37,   0
HJ 1590 data  40,  27,   0,  40,  12,   0,   0,  40,  46,   0,   0,  40
KG 1600 data  47,  49,  47,  48,  22,  53,  49,  47,  48,   3,  47,  49
CF 1610 data  47,  48,  55,  47,  54,  47,  31,  29,  30,  31,  29,  30
EC 1620 data  51,  29,  50,  31,  29,  30,   4,  29,  30,  29,  30,  16
CE 1630 data  29,  52,  31,  29,  30,  19,  17,  19,  17,  20,  26,  17
OO 1640 data  21,  19,  17,  20,   8,  17,  17,  20,  14,  17,  17,  20
EE 1650 data  18,  43,  18,  43,  24,  25,  43,  33,  18,  43,  24,   5
DI 1660 data  43,  43,  24,  45,  43,  43,  24,  35,  36,  66,  69,  32
GJ 1670 data  32,  36,  67,  69,  32,  32,  32,  36,  66,  69,  32,  32
KI 1680 data  32,  65,  69,  32,  32,  32,  36,  82,  69,  32,  32,  32
JL 1690 data  32,  40,  36,  66,  44,  88,  41,  40,  36,  67,  41,  69
KP 1700 data  32,  36,  66,  44,  88,  69,  32,  36,  66,  44,  89,  69
CM 1710 data  32,  69,  32,  32,  32,  32,  32,  36,  67,  44,  88,  69
CP 1720 data  32,  36,  67,  44,  89,  69,  32,  40,  36,  66,  41,  44
LH 1730 data  89,  67,  89,  67,  76,  69,  32,  67,  79,  85,  78,  84
MC 1740 data  69,  82,  32,  86,  48,  46,  48,  50,  44,  32,  66,  89
KE 1750 data  32,  68,  65,  86,  73,  68,  32,  83,  65,  78,  78,  69
MN 1760 data  82,  13,   0,  13,  83,  84,  65,  82,  84,  32,  65,  68
OD 1770 data  68,  82,  69,  83,  83,  47,  60,  67,  82,  62,  32,  58
CE 1780 data  36,   0,  13,  69,  78,  68,  32,  65,  68,  68,  82,  69
ID 1790 data  83,  83,  47,  66,  47,  60,  67,  82,  62,  32,  58,  36
EF 1800 data   0,  13,  69,  88,  69,  67,  85,  84,  69,  32,  73,  78
PH 1810 data  83,  84,  82,  85,  67,  84,  73,  79,  78,  83,  63,  40
DB 1820 data  89,  47,  78,  47,  60,  67,  82,  62,  41,  32,  58,   0
CH 1830 data  13,  80,  82,  73,  78,  84,  32,  84,  82,  65,  67,  69
LF 1840 data  32,  40,  89,  47,  78,  47,  60,  67,  82,  62,  41,  58
IP 1850 data   0,  13,  65,  32,  32,  88,  32,  32,  89,  32,  32,  83
HE 1860 data  84,  32,  83,  80,  13,   0,  13,  69,  82,  82,  79,  82
ON 1870 data  58,  32,  85,  78,  75,  78,  79,  87,  78,  32,  79,  80
NK 1880 data  67,  79,  68,  69,  32,  65,  84,  32,  36,  70,  69,  68
AJ 1890 data  67,   0,  13,  84,  79,  84,  65,  76,  32,  35,  32,  79
IM 1900 data  70,  32,  67,  89,  67,  76,  69,  83,  32,  85,  83,  69
JF 1910 data  68,  58,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32
LD 1920 data  32,   0,  32,  61, 199, 201,   0, 240,   1,   0,  32,  72
KD 1930 data 203, 169,  13,  32, 210, 255, 173,  28, 192, 240,   7,  32
NE 1940 data 175, 198, 201,   1, 240, 112,  32, 208, 198, 173,  25, 192
CA 1950 data 240,   7,  48, 102, 173,  28, 192, 240,  97,  32,  14, 199
IN 1960 data  32, 184, 197, 208,   5,  32, 185, 198, 176, 207,  32, 242
HM 1970 data 197, 173, 104, 193,  41,   2, 240,  47,  32, 155, 198, 208
IL 1980 data  22,  32, 226, 196, 234, 234, 234,  32, 251, 198, 186, 142
GF 1990 data  17, 192,  32,  29, 199,  32,  43, 199,  56, 176,  20, 152
MG 2000 data  24,  42, 168, 185, 150, 193, 141, 193, 196, 200, 185, 150
BA 2010 data 193, 141, 194, 196,  32,   0,   0, 173, 104, 193,  41,   1
KN 2020 data 240,   3,  32,  31, 202,  32, 174, 203, 173, 104, 193,  41
CG 2030 data   4, 240,   3,  32,  61, 200, 169,   0, 240, 132,  32, 171
PO 2040 data 202,   0,  32,  25, 199,  32,  47, 199,  32, 236, 198,  96
KM 2050 data 104, 168, 104, 170, 165, 251, 208,   2, 198, 252, 165, 252
```

```
GC 2060 data  72, 198, 251, 165, 251,  72, 173, 157, 196, 133, 251, 173
IJ 2070 data 158, 196, 133, 252, 206,  17, 192, 206,  17, 192, 138,  72
LK 2080 data 152,  72,  96, 104, 168, 104, 170, 104, 133, 251, 104, 133
LN 2090 data 252, 230, 251, 208,   8, 230, 252, 238,  17, 192, 238,  17
HH 2100 data 192, 138,  72, 152,  72,  96, 173, 157, 196, 133, 251, 173
KM 2110 data 158, 196, 133, 252,  96, 173, 157, 196, 141,  78, 197, 173
CK 2120 data 158, 196, 141,  79, 197,  32,  25, 199,  32,  47, 199, 162
HF 2130 data   0, 173, 255, 255, 157, 111, 193, 238,  78, 197, 232, 224
MJ 2140 data   2, 208, 242,  32,  43, 199,  96, 173,  22, 192, 240,  10
KL 2150 data 173,  23, 192, 133, 251, 173,  24, 192, 133, 252,  96, 104
CA 2160 data 170, 104, 168,  24, 169,   2, 101, 251, 133, 251, 144,   2
AG 2170 data 230, 252, 165, 252,  72, 165, 251,  72, 173,  16, 192,   9
LH 2180 data   4, 141,  16, 192,  72, 173, 254, 255, 133, 251, 173, 255
AO 2190 data 255, 133, 252, 152,  72, 138,  72,  96, 104, 168, 104, 170
FK 2200 data 104, 141,  16, 192, 104, 133, 251, 104, 133, 252,  24, 169
NO 2210 data   3, 109,  17, 192, 141,  17, 192, 138,  72, 152,  72,  96
GA 2220 data 172, 156, 196, 169, 234, 141, 157, 196, 141, 158, 196, 185
MA 2230 data  34, 192, 240,  41, 141,  18, 192,  41, 127,  74,  74,  74
KK 2240 data 168, 190,  34, 193, 138,  72, 240,  11, 160,   1, 177, 251
EP 2250 data 153, 156, 196, 200, 202, 208, 247, 104,  24, 105,   1, 101
JB 2260 data 251, 133, 251, 144,   2, 230, 252, 169,   1,  96, 160,   0
CF 2270 data 173, 104, 193,  41,   2, 240,  20, 173,  18, 192,  48,  12
KI 2280 data  41, 120, 201,  32, 208,   9,  32,  47, 198,  56, 176,   3
HG 2290 data  32, 115, 198, 173,  18, 192,  41,   7,  24, 121,  19, 192
GG 2300 data  24, 248, 109,  33, 192, 141,  33, 192, 162,   3, 169,   0
FC 2310 data 125,  29, 192, 157,  29, 192, 202,  16, 245, 216,  96, 160
AL 2320 data   0, 173, 156, 196, 141,  62, 198, 173,  16, 192,  41, 195
AE 2330 data  72,  40, 240,   6, 140,  22, 192,  56, 176,  44, 200, 140
GC 2340 data  22, 192, 173, 157, 196,  48,  19,  24, 101, 251, 141,  23
EB 2350 data 192, 144,   1, 200, 169,   0, 101, 252, 141,  24, 192,  56
FK 2360 data 176,  16,  24, 101, 251, 141,  23, 192, 176,   1, 200, 165
AF 2370 data 252, 233,   0, 141,  24, 192,  96, 173,  18, 192, 172,  15
NE 2380 data 192,  41, 120, 201,  80, 208,   3, 172,  14, 192, 201,  96
JL 2390 data  24, 240,   8, 152, 109, 157, 196, 144,   8, 176,   6, 152
OG 2400 data 174, 157, 196, 117,   0, 160,   0, 144,   1, 200,  96, 160
FI 2410 data  13, 173, 156, 196, 217,  47, 193, 240,   7, 136,  16, 248
IC 2420 data 169,   0, 240,   2, 169,   1,  96, 174,  27, 192, 172,  26
CN 2430 data 192,  32, 161, 203,  96, 174, 184, 193, 172, 185, 193,  32
NP 2440 data  25, 200, 173, 217, 193, 174, 216, 193,  32, 121, 202,  32
JG 2450 data 171, 202,  56,  96, 165, 203, 201,   4, 208,   8, 169,   4
IO 2460 data  13, 104, 193, 141, 104, 193, 160,   0, 177, 251, 141, 156
PF 2470 data 196, 208,   4, 200, 140,  25, 192,  96, 174,  14, 192, 172
CL 2480 data  15, 192, 173,  16, 192,  72, 173,  13, 192,  40,  96, 141
JL 2490 data  13, 192,   8, 104, 141,  16, 192, 142,  14, 192, 140,  15
BO 2500 data 192, 169,   0,  72,  40,  96, 165, 251, 141, 105, 193, 165
AE 2510 data 252, 141, 106, 193,  96, 160,   7, 208,   2, 160,   3, 162
NH 2520 data   3, 181, 251, 153, 107, 193, 136, 202,  16, 247,  96, 160
OM 2530 data   7, 208,   2, 160,   3, 162,   3, 185, 107, 193, 149, 251
PC 2540 data 136, 202,  16, 247,  96, 173, 207, 193, 174, 206, 193,  32
KL 2550 data 121, 202, 173, 209, 193, 174, 208, 193,  32, 121, 202, 174
HD 2560 data 193, 193, 172, 192, 193,  32, 140, 202, 192,   0, 240,  65
KI 2570 data 174, 192, 193, 173, 193, 193, 160,   3,  32, 224, 201, 174
HA 2580 data 116, 193, 134, 251, 174, 115, 193, 134, 252, 169,   0, 141
KF 2590 data 104, 193, 141,  25, 192, 173, 211, 193, 174, 210, 193,  32
BC 2600 data 121, 202, 174, 193, 193, 172, 192, 193,  32, 140, 202, 192
CO 2610 data   0, 240,  14, 173,  61, 193, 201,  66, 208,   9, 169,   0
AH 2620 data 141,  28, 192, 240,  28, 240, 119, 141,  28, 192, 174, 192
GJ 2630 data 193, 173, 193, 193, 160,   3,  32, 224, 201, 174, 116, 193
AG 2640 data 142,  26, 192, 172, 115, 193, 140,  27, 192, 173, 213, 193
AI 2650 data 174, 212, 193,  32, 121, 202, 174, 193, 193, 172, 192, 193
DI 2660 data  32, 140, 202, 192,   0, 240,  71, 173,  61, 193, 201,  89
KP 2670 data 208,   8, 173, 104, 193,   9,   2, 141, 104, 193, 173, 215
NJ 2680 data 193, 174, 214, 193,  32, 121, 202, 174, 193, 193, 172, 192
MB 2690 data 193,  32, 140, 202, 169,  13,  32, 210, 255, 192,   0, 240
ON 2700 data  29, 173,  61, 193, 201,  89, 240,   2, 208,   8, 173, 104
AI 2710 data 193,   9,   1, 141, 104, 193, 162,   4, 169,   0, 157,  29
BI 2720 data 192, 202,  16, 250,  48,   2, 169,   1,  96, 142,  50, 200
MO 2730 data 140,  51, 200, 174, 182, 193, 173, 183, 193, 160,   2,  32
JH 2740 data 142, 201, 168, 162,   2, 136, 185, 115, 193, 157, 255, 255
PI 2750 data 202, 208,   2, 162,   4, 136,  16, 242,  96, 169,  62,  32
NM 2760 data 210, 255, 174, 193, 193, 172, 192, 193,  32, 140, 202, 162
HG 2770 data   0, 189,  61, 193, 201,  32, 208,   4, 232, 136, 208, 245
CO 2780 data 201,  67, 208,  12, 173, 104, 193,  41, 251, 141, 104, 193
PJ 2790 data 169,   0, 240,  71, 201,  81, 208,  10, 173,  25, 192,   9
EE 2800 data 129, 141,  25, 192, 208,  57, 201,  82, 208,   3,  32,  72
KN 2810 data 203, 201,  84, 208,  11, 169,   1,  77, 104, 193, 141, 104
BF 2820 data 193,  56, 176,  35, 201,  87, 208,  31, 174, 194, 193, 173
MC 2830 data 195, 193, 160,   3,  32, 224, 201, 173, 115, 193, 141, 149
EJ 2840 data 193, 173, 116, 193, 141, 148, 193, 169,   8,  13, 104, 193
DL 2850 data 141, 104, 193, 169,  13,  32, 210, 255,  96, 169,   0, 170
OJ 2860 data 172, 156, 196, 192,   0, 240,   9, 217,  34, 192, 240,   1
JA 2870 data 232, 136, 208, 247,  96,  32, 181, 200, 169,   0, 133, 254
JC 2880 data 189, 136, 194,  24, 133, 253,   6, 253, 144,   3, 230, 254
HD 2890 data  24, 101, 253, 133, 253, 144,   3, 230, 253,  24, 173, 218
NE 2900 data 193, 101, 253, 133, 253, 173, 219, 193, 101, 254, 133, 254
EM 2910 data  96, 174, 156, 196, 189,  34, 192,  41, 120,  74, 133, 253
JP 2920 data  74,  24, 101, 253, 109, 202, 193, 133, 253, 169,   0, 109
KE 2930 data 203, 193, 133, 254,  96, 160,   0, 162,   0, 177, 253, 200
AF 2940 data 142, 145, 193, 140, 144, 193, 201,  66, 208,   4, 160,   1
HE 2950 data 208,  22, 201,  67, 208,   4, 160,   2, 208,  14, 201,  82
DA 2960 data 208,  79, 174, 178, 193, 173, 179, 193, 160,   2, 208,   6
JN 2970 data 173, 199, 193, 174, 198, 193,  32, 142, 201,  56, 233,   1
GK 2980 data 168, 192,   3, 208,  26, 160,   2, 174, 145, 193, 185, 115
ED 2990 data 193, 157,  71, 193, 232, 200, 192,   4, 208,   2, 160,   0
HH 3000 data 192,   2, 208, 238, 202, 208,  21,  24, 109, 145, 193, 141
CB 3010 data 145, 193, 170, 185, 115, 193, 157,  71, 193, 202, 136,  16
FD 3020 data 246, 174, 145, 193, 172, 144, 193, 208,   7, 201,  69, 240
KD 3030 data   8, 157,  71, 193, 232, 192,   6, 208, 136,  96,  72, 165
PO 3040 data 253, 141, 146, 193, 165, 254, 141, 147, 193, 104, 134, 253
AK 3050 data 133, 254, 169,   0, 141, 125, 193, 152,  10,  72, 170, 136
EB 3060 data 202, 177, 253,  72,  41,  15, 142, 126, 193, 170, 189, 128
BP 3070 data 193, 174, 126, 193, 157, 115, 193, 173, 125, 193, 208,  13
DB 3080 data 238, 125, 193, 104,  41, 240,  74,  74,  74,  74, 202,  16
BP 3090 data 225, 206, 125, 193, 202, 136,  16, 213, 173, 146, 193, 133
HP 3100 data 253, 173, 147, 193, 133, 254, 104,  96, 142, 240, 201, 141
GN 3110 data 241, 201, 169,   1, 141, 125, 193, 152,  74, 170,  72, 185
OB 3120 data 255, 255,  56, 233,  65,  24,  48,   4, 105,  10,  16,   2
EC 3130 data 105,  17, 206, 125, 193, 240,  20, 238, 125, 193, 238, 125
NB 3140 data 193,  10,  10,  10,  10,  13, 127, 193, 157, 115, 193, 202
IP 3150 data  24, 144,   3, 141, 127, 193, 136,  16, 210, 104,  96, 165
KE 3160 data 253,  72, 165, 254,  72, 160,  38, 169,  32, 153,  61, 193
DJ 3170 data 136,  16, 250, 169,  36, 141,  61, 193, 174, 180, 193, 172
HA 3180 data 181, 193,  32,  25, 200,  32, 201, 200, 160,   2, 177, 253
ME 3190 data 153,  67, 193, 136,  16, 248,  32, 245, 200,  32,  17, 201
NP 3200 data  32,  34, 203, 174, 186, 193, 173, 187, 193, 160,   3,  32
LK 3210 data 142, 201, 170, 202, 189, 115, 193, 157,  94, 193, 202,  16
NJ 3220 data 247, 173, 193, 193, 174, 192, 193,  32, 121, 202, 104, 133
IC 3230 data 254, 104, 133, 253,  96, 133, 254, 134, 253, 160,   0, 177
NI 3240 data 253, 201,   0, 240,   6,  32, 210, 255, 200, 208, 244,  96
PI 3250 data 165, 253,  72, 165, 254,  72, 134, 254, 132, 253, 160,   0
HI 3260 data  32, 207, 255, 201,  13, 240,   5, 145, 253, 200, 208, 244
DA 3270 data 104, 133, 254, 104, 133, 253,  96, 174, 188, 193, 173, 189
FN 3280 data 193, 160,   5,  32, 142, 201, 170, 202, 189, 115, 193, 157
CA 3290 data  71, 196, 202,  16, 247, 173, 221, 193, 174, 220, 193,  32
LK 3300 data 121, 202,  32, 207, 255,  96, 173,   4,   3, 141, 246, 202
PO 3310 data 173,   5,   3, 141, 253, 202, 173, 190, 193, 141,   4,   3
FG 3320 data 173, 191, 193, 141,   5,   3,  96, 141,  13, 192, 140,  15
LK 3330 data 192, 142,  14, 192,   8,  40, 141,  16, 192, 169,   0, 133
PA 3340 data 251, 141,   4,   3, 169,   0, 133, 252, 141,   5,   3,  32
OA 3350 data 113, 199, 201,   0, 240,   1,   0, 169,  82, 141,   0,   2
AE 3360 data 169,  85, 141,   1,   2, 169,  78, 141,   2,   2, 169,   0
HC 3370 data 141,   3,   2,  76,  90, 196, 173, 197, 193, 174, 196, 193
AM 3380 data 160,   5,  32, 142, 201, 162, 255, 160,   0, 232, 189, 115
AN 3390 data 193, 153,  79, 193, 200, 138,  74, 144, 244, 169,  32, 153
NI 3400 data  79, 193, 200, 224,  11, 208, 234,  96, 173, 223, 193, 174
CP 3410 data 222, 193,  32, 121, 202,  32,  34, 203, 169,  13, 141,  94
IN 3420 data 193, 169, 145, 141,  95, 193, 169,   0, 141,  96, 193, 173
BG 3430 data 201, 193, 174, 200, 193,  32, 121, 202, 174, 193, 193, 172
DH 3440 data 192, 193,  32, 140, 202, 162,   0, 160,   0, 185,  61, 193
JB 3450 data 201,  13, 240,  11, 201,  32, 240,   4, 157,  61, 193, 232
BC 3460 data 200, 208, 238, 174, 192, 193, 173, 193, 193, 160,   9,  32
LJ 3470 data 224, 201, 170, 189, 115, 193, 157,  13, 192, 202,  16, 247
GF 3480 data  96, 169,   0, 196, 251, 208,   6, 228, 252, 208,   2, 169
JJ 3490 data   1,  96, 173, 104, 193,  41,   8, 240,  23, 172, 148, 193
NJ 3500 data 174, 149, 193,  32, 161, 203, 201,   0, 240,  10, 169, 247
AE 3510 data  45, 104, 193,   9,   4, 141, 104, 193,  96, 162,  73, 160
GH 3520 data   0,  56, 185, 150, 193, 237,   9, 192, 153, 150, 193, 200
MA 3530 data 185, 150, 193, 237,  10, 192, 153, 150, 193, 136,  24, 185
OM 3540 data 150, 193, 109,  11, 192, 153, 150, 193, 200, 185, 150, 193
LK 3550 data 109,  12, 192, 153, 150, 193, 200, 202, 202,  16, 210, 173
LD 3560 data  11, 192, 141,   9, 192, 173,  12, 192, 141,  10, 192,   0
```

# RS-232 Hardcopy

## *Talking to serial printers*

**by Joseph Buckley**

I am a member of the tiny minority of Commodore users who happen to have an RS-232 serial printer. Since it seems to me that 99.5% of the software written ignores this possibility, I usually find myself either being forced to customize the software, or just doing without.

For some time I have wanted a program that would allow me to redirect the printer output of both BASIC and assembly programs from device #4 to device #2 transparently, without resorting to altering the program. My first approach was to patch into the vector IBSOUT ($0326) and intercept the output there, but I never could figure out why I would always get a '?device not present' error.

Last August, the sysop of the Programmer's SIG on Quantum Link announced a contest for writing articles to be placed online. This was the perfect excuse to finally get the program running. I sat down and worked on puzzling it out. After what couldn't have been more than two minutes the answer just unfolded before me: My mistake was intercepting the vector IBSOUT, when I should have been intercepting the vector IOPEN.

Page $03 in the Commodore 64 holds the indirect vectors for the operating system I/O routines as a point for patches to get hold of, and modify, those routines. What must be done in this case is to alter the vector IOPEN ($031A) to intercept all calls that will OPEN to device #4, the serial port printer, and open instead device #2, the RS-232 port.

At this point, the call to OPEN will check FA ($BA) to see which physical device is to be used; if it is not #4, it continues on its merry way. If it is #4, it will change FA from $04 to $02, the RS-232 device. It will also set the baud rate at $0293. One other necessary change is to modify the RS-232 output buffer pointer, ROBUF ($F9), to point to an unused block of 256 bytes of free memory. Now we can actually open the logical file as if no change had been made. When finished, we will return to the caller.

The assembly code to redirect the output is as follows:

```
patch1   lda   $ba       ; check current device
         cmp   #$04      ; is it a printer?
         bne   patch1a   ; no? then ignore patch
         lda   #$02      ; yes, redirect to RS-232 port
         sta   $ba       ; new device
         lda   #$07      ; baud rate (600)
         sta   $0293     ; set baud
         lda   #<buff    ; buffer address
         ldx   #>buff    ; may change
         sta   $f9
         stx   $fa
patch1a  jsr   openfl    ; open the logical file
         clc
         rts             ; done
```

The address of OPENFL is loaded with the initial value of IOPEN. This will allow multiple patches to run concurrently.

While you will now no longer generate a '?device not present' error if there is no device #4, having no device #2 will not cause an error either. The computer will carry on whether or not your printer is powered up.

Most RS-232 printers will also need linefeeds in addition to each carriage return sent to it. (One thing improperly stated in the Reference Guide is that a logical file number which is greater than 127 adds a linefeed. While this is true for BASIC, ML calls to $FFD2 make no provision for this.) To handle this problem, another patch is needed at IBSOUT ($0326). This will add any linefeeds as well as PETASCII to true ASCII conversions if needed.

```
patch2  sta   temp      ; save A
        lda   $9a       ; check current device
        cmp   #$02      ; is is 2?
        bne   out2      ; no, ignore patch
        lda   temp      ; restore A
        cmp #$0d        ; is it a C/R?
        bne patch2a     ; no, skip LF
        lda #$0a        ; LF ($00 / no LF)
        jsr prntit      ; send LF
        lda #$0d        ; the old C/R patch2a
        sec             ; CLC for PetASCII
        bcc   out1      ; send TrueASCII
        cmp   #$41
        bcc   patch2b   ; <a?, yes, skip
        cmp   #$5b
        bcs   patch2b   ; >z?, yes, skip
        clc
        adc   #$20      ; make TrueASCII
        bne   out1      ; exit patch2b
        cmp   #$c1
        bcc   patch2c   ; <A?, yes, skip
        cmp   #$db
        bcs   patch2c   ; >Z?, yes, skip
        sec
        sbc   #$80      ; make TrueASCII
        bne   out1      ; exit patch2c
        cmp   #$14      ; CBM delete?
        bne   out1      ; no, exit
        lda   #$08      ; make TrueASCII backspace
out1    sta   temp      ; save A
out2    lda   temp      ; restore A
        jmp   prntit    ; print/exit
```

The address of PRNTIT is loaded with the initial value of IBSOUT ($0326) to allow for other patches, while TEMP is any free byte of RAM.

The SEC at PATCH2A acts as a flag for the ASCII conversion to follow. If no conversion is needed, then either delete the code, or put a CLC in its place to branch around the conversion.

What follows is the BASIC loader for the code. It is written to be relocatable by changing the value of AD. You will need approximately 352 bytes for this routine, since the 256 byte RS-232 output buffer resides at the end of the code. It has also been commented so that it can be easily customized to suit the application.

Once run, you merely load another program under it and all output to device #4 will be redirected to device #2. As long as the new program doesn't do a major rework to the system, or use the two bytes $F9-FA, and you can find a place to store the code and buffer, you'll be all right. As usual, RUN/STOP-RESTORE will reset the indirect vectors to their default values and disable the patches.

By the way, I managed to place second in the article contest!

```
JE 10 rem*   rs232 printer driver for c64
JE 20 rem*   redirects data for device #4 to
DN 30 rem*   rs232 port (device #2)
MJ 40 :
JL 100 ad=49152
GE 110 for t=0 to 94: read x: poke 49152+t,x: next
MO 120 :
GJ 130 rem set jmp address for open
PF 140 ol=peek (794): oh=peek (795)
LA 150 poke ad+24,ol: poke ad+25,oh
JH 160 ah=int(ad/256): poke 794,ad-(256*ah): poke 795,ah
OB 170 :
HL 180 rem set rs-232 output buffer address
FE 190 bh=int((ad+96)/256): bl=ad+96-(256*bh)
MA 200 poke ad+16,bl: poke ad+18,bh
GE 210 :
GD 220 rem set baud rate
NG 230 poke ad+11,7
EG 240 :
AH 250 rem set temporary storage
DH 260 sh=int((ad+95)/256): sl=ad+95-(sh*256)
BK 270 poke ad+29,sl: poke ad+30,sh
DL 280 poke ad+38,sl: poke ad+39,sh
HN 290 poke ad+87,sl: poke ad+88,sh
LM 300 poke ad+90,sl: poke ad+91,sh
KK 310 :
CJ 320 rem add linefeed 10 yes/0 no
ND 330 poke ad+45,10
IM 340 :
FF 350 rem set ascii conversion flag (56 yes/24 no)
BG 360 poke ad+51,56
GO 370 :
LJ 380 rem set jmp address for print
FD 390 pl=peek (806): ph=peek (807)
BC 400 poke ad+47,pl: poke ad+48,ph
JD 410 poke ad+93,pl: poke ad+94,ph
GI 420 ph=int((ad+28)/256): pl=ad+28-(256*ph)
BG 430 poke 806,pl: poke 807,ph
MC 440 :
HN 1000 data 165, 186, 201,   4, 208,  17, 169,   2
MC 1010 data 133, 186, 169,   7, 141, 147,   2, 169
PI 1020 data  96, 162, 192, 133, 249, 134, 250,  32
OG 1030 data  74, 243,  24,  96, 141,  95, 192, 165
ED 1040 data 154, 201,   2, 208,  52, 173,  95, 192
BA 1050 data 201,  13, 208,   7, 169,  10,  32, 202
HE 1060 data 241, 169,  13,  56, 144,  32, 201,  65
ND 1070 data 144,   9, 201,  91, 176,   5,  24, 105
GE 1080 data  32, 208,  19, 201, 193, 144,   9, 201
HD 1090 data 219, 176,   5,  56, 233, 128, 208,   6
GJ 1100 data 201,  20, 208,   2, 169,   8, 141,  95
IM 1110 data 192, 173,  95, 192,  76, 202, 241
```

# StarCart

## *Computing the relative positions of stars*

**by Stephen Shervais, Jr.**


For many applications, and for most amateur purposes, the standard astronomical coordinate system, Right Ascension and Declination, is ideal. Since the sky appears to be a sphere rotating about a pole, polar coordinates exactly match the view, and portray relative positions of stars with a minimum of fuss and distortion. Even the quirky modifications which astronomers have made to the simplest polar coordinate system - using hours instead of degrees of longitude and measuring 90 degrees each way from the equator instead of 360 degrees from one pole - even these have good practical historical reasons behind them. And when our concerns become three-dimensional and the positions of stars relative to the Sun become important, Right Ascension, Declination, and Distance still give us all the information we need. The system only becomes unwieldy when we need to discuss the relative positions of stars to each other. Then we find ourselves trying to compare two distances and four angles and wondering why we have a headache.

There are a number of reasons why we might wish to look at relative star positions. We might be interested in the distribution of different types of stars, or be trying to decide just where the Sun lies in relation to the rest of the Orion Arm of the Galaxy. We might be building a three-dimensional map of the Solar Neighborhood, or even a Galactic Orrery. Fortunately, our old friend the Cartesian coordinate system (X, Y, and Z axes at right angles to each other) is available to help compare the distances between the stars. Using Cartesian coordinates, calculating distances is only a matter of solving the Pythagorean Theorem: $D^2=X^2+Y^2+Z^2$. The hard part is getting from astronomical to Cartesian without going mad.

Enter the computer. The formulae for converting polar to Cartesian coordinates are relatively simple, and available in any good book on analytic geometry:

X=Distance * Sin(Right Ascension) * Cos(Declination)
Y=Distance * Sin(Right Ascension) * Sin(Declination)
Z=Distance * Cos(Right Ascension)

There are three pitfalls to be avoided when getting your computer to do the job for you. First, you have to be sure to convert the astronomical data to true polar coordinates. Second, and simultaneously, you must remember that your computer works in radians, not degrees. Finally, you must be careful in data entry formats so that your program is converting the numbers you think it is converting, and in the direction you want them to go.

The following program will allow you to create a file of Cartesian coordinates for any (reasonable) number of stars, then print out either the coordinates themselves, or the distances from each star to all the others. Line 200 starts everything off by defining a function to convert degrees and minutes to decimal degrees (or hours and minutes to decimal hours), while lines 220-310 serve as the menu function. Lines 340-510 take in the standard astronomical information, check to see that it is correct (more precisely, that it is what you intended to type), and convert the positional data to radians. Note that the input format requires a decimal to separate the hours or degrees from the minutes; those who use decimal degrees or hours must modify line 200 so that lines 500 and 510 will give the correct answer.

The program then jumps successively to two subroutines which convert the coordinates to Cartesian (lines 570-630) and then output a sequential file to disk (650-740).

You now have a sequential file on your disk which contains both the original input data and the Cartesian coordinates. If you rerun the program and ask for a coordinate output, lines 760-950 will read the file from the disk, and lines 970-1100 will print it out. The subroutine on lines 1120-1320 is the interesting one. It takes the data from the file, computes the distance between each star and every other star, and prints out the result. Be warned, the length of the printout grows very rapidly as you add stars to the file. Users might want to add a line after 1110 that asks for a maximum distance (i.e. only print stars closer to each other than five light years), and put an IF/THEN statement into line 1210 so the program will respond accordingly.

There are limitations to the program as written, and users may wish to write additional subroutines to overcome them. One limitation is that you have to tell the program how many records to read, which means you must either remember the number or remember where you wrote it down. This presents the reader with the opportunity to write a subroutine that will read the file and count how many records are there. Another limitation is that there is no provision for inserting additional records; you must start over each time. Finally, the cooordinate system is aligned with the plane of the Ecliptic (defined by the orbit of Earth around the Sun), a direction which has almost no significance in the greater scheme of things. A truly fun project would be the addition of a subroutine which would convert the astronomical (ecliptic) coordinates to the galactic latitude and longitude system. The key problem there is to find a way to resolve the quadrant ambiguities which result. This is left as an exercise for the reader.

**StarCart 1.0**: *Astronomical to Cartesian coordinate conversion program for determining relative positions of stars.*

```
CE  100 rem starplot 1.0
HK  110 rem copyright @ 1987
ID  120 rem by stephen shervais jr
FI  130 rem 4868 langer ln
FB  140 rem woodbridge, va, 22193
JC  150 rem compuserve 72060,573
DB  160 print"this program is designed to compute"
FA  170 print"the distance between stars near the sun"
FL  180 print"in cartesian coordinates aligned with"
NG  190 print"the plane of the ecliptic":print
IC  200 def fna(q)=(int(q)/57.296)+((q-int(q))/60/57.296)
KF  210 rem **** input or output *****
GM  220 print"do you wish to create a new file or print out an old one?"
IB  230 print
DD  240 print"1. new file"
MO  250 print"2. old file: cartesian coordinates"
OP  260 print"3. old file: distances"
AE  270 print
LE  280 get a1$: if a1$="" then 280
IO  290 if a1$="1" then gosub 340
BC  300 if a1$="2" then gosub760: gosub970
BO  310 if a1$="3" then gosub760: gosub1120
AE  320 end
AL  330 rem **** input new file ****
EB  340 input"total number of star systems";s1:s1=s1-1
DG  350 dim s$(s1), s(s1,8)
DG  360 for t=0 to s1
MA  370 print"{rvs off}system";t+1
BM  380 input"system name     ";s$(t)
IN  390 input"right ascension ";s(t,0)
CO  400 input"declination     ";s(t,1)
BI  410 input"distance, ly    ";s(t,4)
KN  420 rem **** error check ****
AO  430 print
KI  440 print"check:     name "; s$(t)
EB  450 print"           r.a. "; s(t,0)
DG  460 print"           dec  "; s(t,1)
JK  470 print"           dist "; s(t,4)
DE  480 input"is this correct (y/n)  y{left}{left}{left}";a2$
NJ  490 if a2$<>chr$(89) then 370
MK  500 s(t,2)=fna(s(t,0))*15
PL  510 s(t,3)=fna(s(t,1))
CI  520 gosub 570:rem *** xyz coords ***
CB  530 next t
PP  540 gosub 650:rem ** output to file **
CE  550 return
MB  560 rem ** ecliptic cartesian conv **
CC  570 x=s(t,4)*sin((90/57.296)-s(t,3))*cos(s(t,2))
DO  580 s(t,5)=x
JD  590 y=s(t,4)*sin((90/57.296)-s(t,3))*sin(s(t,2))
KP  600 s(t,6)=y
FO  610 z=s(t,4)*cos((90/57.296)-s(t,3))
BB  620 s(t,7)=z
CJ  630 return
EO  640 rem *** output to disk ***
BO  650 input "new file name?";a3$
AC  660 open14,8,14,"0:"+a3$+",s,w"
JJ  670 for t=0 to s1
PL  680 print#14,s$(t)
EH  690 for i=0 to 7
CC  700 print#14,s(t,i)
FK  710 next i
AN  720 next t
KE  730 print#14:close14
AA  740 return
BE  750 rem *** input from disk ***
OG  760 input "old file name ";a3$
PL  770 print"do you want to output to screen,"
OI  780 print"or printer?":print
KB  790 print"1. output to screen"
JB  800 print"2. output to printer"
KF  810 get b1$: if b1$="" then 810
LI  820 if b1$<>"1" and b1$<>"2" then 810
AH  830 print
BE  840 input "number of star systems";s1:s1=s1-1
FF  850 if s<0 then return
BG  860 dim s$(s1), s(s1,8)
KM  870 open10,8,10,"0:"+a3$+",s,r"
LG  880 for t=0 to s1
JF  890 input#10,s$(t)
GE  900 for i=0 to 7
ML  910 input#10,s(t,i)
HH  920 next i
CK  930 next t
MP  940 print#10:close10
CN  950 return
GA  960 rem *** ecliptic output ***
PI  970 open4,3: if b1$="2" then close4: open4,4
PM  980 for t=0 to s1
KK  990 print#4
AP  1000 print#4,s$(t)
AF  1010 print#4,"right ascension ";s(t,0);"hours.minutes"
OG  1020 print#4,"     declination ";s(t,1);"degrees.minutes"
IE  1030 print#4,"       distance= ";s(t,4);"light years"
JG  1040 print#4,"              x= ";s(t,5);"light years"
HH  1050 print#4,"              y= ";s(t,6);"light years"
FI  1060 print#4,"              z= ";s(t,7);"light years"
GI  1070 if b1$="1" then gosub 1340
ID  1080 next t
OP  1090 print#4:close4
IG  1100 return
NI  1110 rem *** distance output ***
FC  1120 open4,3: if b1$="2" then close4: open4,4
JI  1130 print#4,"distance from sun to ":print
PG  1140 for t=0 to s1
LD  1150 print#4,s$(t);tab(25-len(s$(t)));s(t,4)
MN  1155 if b1$="1" then gosub 1440
II  1160 next t
OE  1170 print#4:close4
JA  1180 print:input"maximum trip distance ";d2:print
LG  1190 open4,3: if b1$="2" then close4: open4,4
IL  1200 for t=0 to s1-1
FM  1201 gosub 1470
LA  1210 print#4,"distance from ";s$(t);" to:":print:b2=0
NA  1220 for u=t+1 to s1
CP  1230 x1=s(t,5)-s(u,5):x2=x1*x1
JA  1240 y1=s(t,6)-s(u,6):y2=y1*y1
AC  1250 z1=s(t,7)-s(u,7):z2=z1*z1
MH  1260 d1=sqr(x2+y2+z2)
EF  1270 if d1<d2 then print#4,s$(u);tab(25-len(s$(u)));d1
JF  1280 if b1$="1" then gosub 1440
NA  1290 next u
AF  1300 print:next t
KN  1310 print#4:close4
EE  1320 return
OE  1330 rem *****screen counter****
CK  1340 b2=b2+8
MM  1350 if b2=24 then gosub 1370
MG  1360 return
KD  1370 print"hit any key to continue{up}"
EP  1380 get b2$:if b2$="" then 1380
OJ  1390 b2=0: return
KO  1440 b2=b2+1
NC  1450 if b2=22 then gosub 1470
AN  1460 return
OJ  1470 print"hit any key to continue{up}"
MF  1480 get b2$:if b2$="" then 1480
GF  1490 print:b2=0: return
```

# Disabling "i0" on the 1581

## *A peek at the vectored operating system*

by M. Garamszeghy

One of the nice features of the 1581 is its ability to use subdirectories (or, more correctly, disk partitions) to divide the large 800K disk space into smaller, more manageable chunks. However, this feature does not work with all commercial software (such as various versions of PaperClip, Pocket Writer, etc., on both the C64 and C128) because of the annoying habit of such software to log-in the drive with an "i0" command before reading or writing a document file. On the 1581, the "i0" command is not only superfluous for disk log-ins, but will also reset the directory partition back to the normal or 'root' directory area, thus negating any previously made partition selection.

Fortunately, there is a way to de-activate the "i0" command without interfering with other operations of the 1581, thus giving full partition support for virtually all software. The technique outlined below can be extended to other operating system functions such as VALIDATE, SCRATCH and NEW for example.

*You can trap the vectors and replace the code in the drive ROMs with your own code...*

The operating system of the 1581 drive is unusual amongst Commodore disk drives in that the major functions are accessed indirectly through RAM vectors. This means that you can trap the vectors and replace the code in the drive ROMs with code of your own devising. The ample RAM and buffer space of the 1581 also favour such custom programming.

On the 1581, operating system calls are made via a two step jump. The various routines are normally accessed by JSRing or JMPing to a table of addresses beginning at $FF00 of the disk drive's ROM. Each entry in the table consists of an indirect JMP (*xxxx*) instruction, where *xxxx* is an address of a location in the drive's RAM that contains the real address for the routine to be executed. This address, or vector, normally points back to a ROM routine. However, because the ultimate execution address is stored in RAM, it can be easily changed to point to new or custom code.

Table 1 is a list of some of the more important of the 1581's vectored operating system calls and the normal memory locations associated with each. The purpose of each of the functions is described in detail beginning on page 108 of the 1581 user's manual along with the remainder of the operating system calls. To disable any of these commands, it is only necessary to point the vector to a simple RTS instruction somewhere in the drive's ROM. One such convenient location is $807B. Remember that spot - it is all that is required to deactivate any system function and can be very useful.

Back to our original problem. The vector for the "i0" command is at $198. To de-activate the command, a simple memory write is all that is required. After opening the command channel on the required disk drive as logical file 15 (e.g. OPEN 15,8,15 for a device 8 drive), type in:

```
print#15,"m-w";chr$(152);chr$(1);
   chr$(2);chr$(123);chr$(128)
```

This sequence of bytes does a memory write to the RAM in the disk drive (similar to a POKE in the computer) that changes the address vector at $198 to point to the magical RTS instruction mentioned above. Now when your application software issues an "i0" command to the drive, it is just ignored, and your previously selected partition remains selected.

**Note:** *The techniques described here will only work with the 1581 drive and will probably cause other drives to crash or, worse, trash your disks.*

To restore the operation of the "i0" command, you need only issue the drive reset command:

```
print#15,"ui"
```

This automatically resets all RAM vectors to their default values (unless of course, you have intercepted the "ui" vector, in which case it will do nothing). Note that this memory write method must be applied prior to starting up your other software because it is very difficult, if not impossible, to send the required chr$ values to the disk drive from within a commercial program, even one containing a sophisticated DOS wedge.

It will also be reset by a general system reset, such as pressing the reset button on a C128.

To change DOS partitions from within an application program, the application must have a DOS wedge capable of sending any user specified command string to the disk drive. The command to select a disk partition is:

```
/0:<partitionname>
```

where <partitionname> is the name of the disk partition that you wish to select. It will appear in the directory with a file type of CBM. Note that to access it as a sub-directory partition, the partition must first have been formatted as a DOS partition when it was initially created. To return to the normal or 'root' partition, the command is:

```
/0
```

with no partition name specified.

As with all computer programming, there is more than one solution to the problem. This next one uses a DOS ampersand utility file loader to perform the same task in a slightly more elegant fashion. (The DOS ampersand file is a special type of USR file which contains a machine language program to be executed inside the disk drive.) To activate this program, all you need to do is send:

```
&0:i0-off
```

over the disk command channel to the 1581 drive. This is very easy to do with the DOS wedges available in most commercial wordprocessing programs etc. (For example, with Pocket Writer 128, to get to the DOS wedge, press the Commodore logo key then the 'c' key. You can then type in any disk command and send it to the drive. Other programs have similar capabilities.) To restore the "i0" command, send the "ui" command string over the command channel as outlined above.

Listing 1 is a BASIC loader which creates the USR file *i0-off*. It works with all Commodore computers capable of using the 1581 drive (i.e. C64, C128, +/4, etc.) Remember to set line 150 to the device number of your 1581 drive. This device number is only used to create the initial file and is not used as part of the file itself.

Obviously, the technique outlined above can be extended to any of the vectored commands. For example, by changing the vector at $01AA you can disable SCRATCH; $01AC can be used to disable the NEW (format) command.

```
AI  100 rem **************************
PP  110 rem   turn off 1581 i0 command
LH  120 rem   by m. garamszeghy
OJ  130 rem **************************
AA  140 :
LP  150 dv=8          : rem 1581 device number
IP  160 open 2,dv,2,"i0-off,u,w"
PC  170 for i=1 to 15
KM  180 read x
CD  190 print#2,chr$(x);
MM  200 next
IK  210 close 2
AF  220 :
NG  230 data 0, 4       : rem execute dos '&' file at $0400 in drive ram
HE  240 data 11         : rem program is 11 bytes long
FF  250 data 169, 123   : rem   lda #$7b
GO  260 data 141, 152, 1: rem   sta $0198      ; reset i0 vector
LL  270 data 169, 128   : rem   lda #$80       ; to point to a
PF  280 data 141, 153, 1: rem   sta $0199      ; rts command
MJ  290 data 96         : rem   rts
NL  300 data 14         : rem   checksum
```

TABLE 1: 1581 System Vectors

| System Function Name | Main Entry Address | RAM Vector at address | Points to code at |
|---|---|---|---|
| JIDLE | FF00 | 0190 | B0F0 |
| JIRQ | FF03 | 0192 | DAFD |
| JNMI | FF06 | 0194 | AFCA |
| JVERDIR | FF09 | 0196 | B262 |
| JINTDRV | FF0C | 0198 | 8EC5 |
| JPART | FF0F | 019A | B781 |
| JMEM | FF12 | 019C | 892F |
| JBLOCK | FF15 | 019E | 8A5D |
| JUSER | FF18 | 01A0 | 898F |
| JRECORD | FF1B | 01A2 | A1A1 |
| JUTLODR | FF1E | 01A4 | A956 |
| JDSKCPY | FF21 | 01A6 | 876E |
| JRENAME | FF24 | 01A8 | 88C5 |
| JSCRTCH | FF27 | 01AA | 8688 |
| JNEW | FF2A | 01AC | B348 |

# C-128 CP/M Plus Memory Map

## *An in-depth investigation...*

**by Miklos Garamszeghy**

*Herne Data Systems, Toronto, Ontario*

The memory map of the C-128 is complicated to say the least. It is even more complicated in CP/M mode because of the variety of RAM, ROM, and input/output chips used by the CP/M operating system. This article attempts, for the first time, to map out the memory configurations used in CP/M mode. It is not claimed to be complete, but is a very useful starting point for your own explorations. The labels for the various memory locations are taken from the CP/M system source code files.

It is difficult to produce a detailed memory map of the CP/M operating system because much of the operating system is RAM based, and therefore subject to quick and easy revisions. There are currently four versions of C-128 CP/M generally available (apart from various beta test versions), as denoted by the dates displayed on boot up or by pressing the F8 key. There are some major differences in the memory maps of each version. For clarity, these are defined here as:

AUG = Version dated 1 Aug 85
DEC = Versions dated 6 Dec 85 or 8 Dec 85
MAY = Version dated 28 May 87

There are also significant differences in the memory map depending on which processor (the Z-80 or 8502) is currently in use. This is important even in CP/M mode because most of the low-level BIOS routines, such as standard serial port operations (some disk operations and all printer output), RAM disk operations, etc. use the 8502 mode.

Z-80 OPERATIONS
---------------

BANK 0 : MMU configuration register value $3f

BANK 0 is the system bank. The CP/M BIOS and BDOS operate primarily in this bank. System calls are made by transient programs via the common memory.

0000 to 0FFF - Z-80 ROM code

1000 to 128F - SYSKEYAREA ; string data for programmable keys. Each key can be defined as a string. Definitions "float" (i.e. you do not have to adjust any pointers elsewhere) and are terminated by a zero byte. Vector at FD0B points to this table.

1290 to 13EF - KEYCODES ; ASCII codes for each key, 4 values for each key (normal, "alpha mode", shift and control) arranged according to key scan code table on pg. 687 of C128 Programmer's Reference Guide. Note that "alpha mode" defaults to uppercase and is toggled on and off by pressing the C= logo key. It is equivalent to a software "caps lock", but is not related to either the <caps lock> or <shift lock> keys. Vector at FD09 points here.

| Value | Meaning |
|---|---|
| 0 | null (equivalent to no key press) |
| 1 to 7f | normal ASCII codes (letters, numbers, symbols, etc) |
| 80 to 9f | key has been programmed as a string, defined in SYSKEYAREA (32 possible "programmable" keys) |
| a0 to af | 80-col character colour (ctrl with number keys on main keyboard) |
| b0 to bf | 80-col background colour (ctrl with number keys on numeric keypad) |
| c0 to cf | 40-col character colour |
| d0 to df | 40-col background colour |
| e0 to ef | 40-col border colour |
| f0 | toggle disk status line on/off (ctrl-run/stop) |
| f1 | system pause (no-scroll key) |
| f2 | track cursor on 40-col screen (ctrl-no-scroll) |
| f3 | move 40-col window left 1 char (ctrl <- is defined as 4 f3's) |
| f4 | move 40-col window right 1 char (ctrl -> is defined as 4 f4's) |
| f5 | unlock MFM disk types (ctrl-home) |
| f6 | select ADM31 emulation mode (ctrl- on numeric keypad) (MAY version only) |
| f7 | select VT100 emulation mode (ctrl+ on numeric keypad) (MAY version only) |
| f8 to fe | not currently defined - reserved for future expansion |
| ff | system cold re-start (control-enter) |

(NOTE: when bit 7 of FD22 (STATENABLE) is on, key codes of $80 and greater are returned without executing special functions as outlined above. FD4C contains a vector to the address of the table of execution addresses of key codes f0 to ff. This vector can be changed to point to your own code.)

13F0 to 13FF - COLORTBL ; logical colour values 00, 11, 22, ... FF for use with (esc-esc-esc) colour value. Vector at FD0D points here.

1400 to 1BCF - SCREEN40 ; pseudo 80-column screen character buffer for 40-column screen
1C00 to 23CF - COLOR40 ; pseudo 80-column screen colour buffer for 40-column screen

2400 BANKPARMBLK ; (system parameters and flags)
2402 CUROFFSET ; position counters used by pseudo 80-col screen
2404 OLDOFFSET ; for 40-col screen
2405 PRTFLG ;

```
2406 FLASHPOS
2408 PAINTSIZE   ; number of rows to move over for 40-col screen shift
                      ($18 or $19)
2409 CHARADR40   ; pointer to current char in pseudo screen RAM
240B CHARCOL40   ; 40-col character position - column     0-79
240C CHARROW40   ;                            - screen row 0-24
240D ATTR40      ; 40-col attribute (character colour)
240E BGCOLOR40   ;   background colour
240F BDCOLOR40   ;   border colour
2410 REV40       ; reverse video flag

2411 CHARADR     ; pointer to current char in 80-col RAM
2413 CHARCOL     ; 80-col character position - column     0-79
2414 CHARROW     ;                            - screen row 0-24
2415 CURRENTATR  ; 80-col attribute
     bit : 7 - 0 = alternate (block graphics) char set
           - 1 = ASCII character set
     6 reverse video 1 = on, 0 = off      2 green     1 = on, 0 = off
     5 underline     "       "            1 blue      "       "
     4 blink         "       "            0 intensity "       "
     3 red           "       "

2416 BGCOLOR80     ; background colour
2417 CHARCOLOR80   ; character colour

2418 PARMBASE      ; pointers to currently active attribute sets
241A PARMAREA80
241D PARMAREA40
2420 BUFFER80COL

2471 KEYBUF        ; buffer for currently pressed key code

2488 CONTROL CODES ; flag for control/shift keys pressed

     Bit      Key Pressed        Bits 0 and 1    Character Mode
     ----------------------      ------------------------------
      2       control key             00    =    lower case
      4       right shift key         01    =    alpha mode
      5       C= key                  10    =    shift
      7       left shift key          11    =    control

2489 MSGPTR        ; pointer to current function key message string

248B OFFSET        ; cursor pointers used by various screen
248C CURPOS        ; printing routines

248E SYSFREQ       ; power line frequency : 0 = 60 Hz, FF = 50 Hz

2600 to 2A40 - BIOS8502  ; 8502 BIOS code
                     (see map of 8502 mode below for description)

2C00 to 2FFF - VICSCREEN ; 40-column video RAM, also appears in hardware
     I/O area and Bank 2.  (Note this is separate from normal C128 40-col
     video RAM at $0400 which is unused in CP/M mode because it is under the
     Z-80 ROM.)


3000 to 3CFF - CCPBUFFER ; CCP.COM hides here during TPA execution


     ;
3C00 BOOTPARM     3C09 BLOCKPTRS    ;
3C02 LDBLKPTR     3C29 INFOBUFFER   ;
3C04 BLKUNLDPTR   3C35 EXTNUM       ; various flags etc. used during cold boot
3C06 BLOCKSIZE    3C36 RETRY        ;
3C07 BLOCKEND     3C77 BOOTSTACK    ;

3D00 BANK0FREE     ; BANK 0 work area can be used by transient programs
     extends to 98FF on Dec/85 and May/87 versions, 9BFF on Aug/85 version.
     Primarily used by CP/M as directory and file buffers. Programmers can
     use, but beware of implications.
```

Operating system areas:

---------------------

| Component | DEC | MAY | AUG |
|---|---|---|---|
| Bank BDOS | 9900 | 9900 | 9C00 |
| Resident BDOS | EA00 | EA00 | EE00 |
| Bank BIOS | C700 | C800 | CA00 |
| Resident BIOS | F000 | F000 | F400 |

MFM disk parameter table (DPT) is located at d876 to da75 in AUG version
d6bd to d8bc in DEC versions
and d860 to da5f in MAY version

The DPT structure is described in "Inside C128 CP/M" (Transactor vol 8/4, pg. 43.) It contains the basic information to allow access to foreign MFM disk types. The DPT can be found by the pointer at FD46 (all versions).

The other major component of accessing disk drives is the disk parameter header or DPH. This is the working area used by the BDOS for actual disk access. When a disk is logged in, the appropriate values are copied from the DPT to the DPH for general use. The drive table address which contains the vectors to the DPH for each logical drive can be found at BIOSBASE+D7. This contains a series of address pointers to the DPHBASE for each logical drive (A: to P:). If a drive is not supported (drives F: to L: and N: to P:), the drive table value is 0. For each drive the $37 byte long DPH has the following format:

```
DPHBASE-A    pointer to the sector write routine for this drive
DPHBASE-8    pointer to the sector read routine for this drive
DPHBASE-6    pointer to the login routine for this drive
DPHBASE-4    pointer to the initialization routine for this drive
DPHBASE-2    physical drive assigned to the logical drive according to values
              listed below under VICDRV.
DPHBASE-1    secondary disk type byte from byte 2 of DPT entry.
DPHBASE      pointer to logical to physical sector skew table (0000 if none)
DPHBASE+2    9 bytes of scratch pad for use by BDOS
DPHBASE+B    media flag 0 if disk logged in, FF if disk has been changed
DPHBASE+C    pointer to DPB values for this drive (contained later in entry)
DPHBASE+E    pointer to CSV scratch pad area (used to detect changed disks)
DPHBASE+10   pointer to ALV scratch pad area (used to keep track of drive
              storage capacity)
DPHBASE+12   pointer to directory buffer control block (BCB)
DPHBASE+14   pointer to data buffer control block
DPHBASE+16   pointer to the directory hashing table (FFFF if not used)
DPHBASE+17   bank for hash table
DPHBASE+18   DPT entry for this drive
DPHBASE+2A   maximum sector number and MFM lock flag (bit 7 on if locked)
DPHBASE+2B   pointer to entry in master MFM DPT table
```

E000 Free memory in Common BANK 0 and 1. Normally used by RSX (resident system extension) programs, as well as operating system extensions and SID, SUBMIT, SAVE, GET, PUT, etc. Can be used by experienced programmers if you are aware of consequences, such as possible crash when using another program.

Extends to E9FF in DEC and MAY versions and
EDFF in AUG version

System Control Block (SCB):

DEC and MAY Versions EF9C to EFFF
AUG Version F39C to F3FF

100 bytes containing basic system variables, located immediately before BIOS jump table, can be read or written with BDOS function 49. Basic parameters detailed in Appendix A of CP/M Plus Programmers Guide from Digital Research.

| Byte Offset | Function |
|---|---|
| 00 - 04 | reserved for various system flags |
| 05 | BDOS version number |
| 06 - 09 | reserved for user determined flags (put your own stuff here) |
| 0A - 0F | reserved for system use |
| 10 - 11 | 16-bit program return code for passing data to chained programs |
| 12 - 19 | reserved for system use |
| 1A | screen width - 1 (set to 79) |
| 1B | current column position on screen (0 to 79) |
| 1C | screen page length (24 lines per screen) |
| 1D - 21 | reserved for system use |
| 22 - 2B | assignment vectors for CP/M's logical I/O devices. 16-bit value used as follows: |

| Bit | Device | |
|---|---|---|
| f | KEYS | (input only) |
| e | 80COL | (output only) |
| d | 40COL | (output only) |
| c | PRT1 | (device 4 serial printer, output only) |
| b | PRT2 | (device 5 serial printer, output only) |
| a | 6551 | (not really supported, it was supposed to be on an external card which was never produced) |
| 9 | RS232 | (input or output) |
| 8 to 0 | not used | |

If a bit is on, then the physical device is assigned to that logical device. Each logical device can have more than one physical device assigned to it and each physical device can be assigned to more than one logical device. Logical devices and their assignment vectors are:

| | |
|---|---|
| 22 - 23 | CONIN |
| 24 - 25 | CONOUT |
| 26 - 27 | AUXIN |
| 28 - 29 | AUXOUT |
| 2A - 2B | LSTOUT |

```
2C      Page mode 0 = display 1 page of data at a time
                  1 = display continuously
        (this is the annoying flag that causes CP/M TYPE command to say
        "press return to continue" after each screenful of data.)
2D      reserved for system use
2E      determines effect of CTRL-H.  0 = backspace and delete
                                     FF = delete and echo
2F      determines effect of delete   0 = delete and echo
                                     FF = backspace and delete
30 - 32 reserved for system use
33 - 34 16-bit console mode flag (default value = 0000):
```

| Bit | Meaning |
|---|---|
| 0 | 0 = return normal status for BDOS function 11<br>1 = CTRL-C only status |
| 1 | 0 = enable CTRL-S CTRL-Q stop scroll/start scroll<br>1 = disable stop/start scroll |
| 2 | 0 = normal console output<br>1 = raw console output, disables tab expansion, and CTRL-P printer echo |
| 3 | 0 = enable CTRL-C program termination<br>1 = disable CTRL-C |
| 8 - 9 | used for RSX's |

```
35 - 36 address of 128 byte scratch pad buffer
37      output string delimiter (normally $)
38      LIST output flag  0 = console output only
                          1 = echo output to printer
39      reserved for system use
3A - 3B pointer to start of SCB
3C - 3D current DMA (disk buffer) address
3E      current drive (0 = A:, 1 = B:, etc.)
3F - 40 BDOS disk info flags
41      FCB flag
43      BDOS function where error occured
44      current user number (0 to F)
45 - 49 reserved for system use
4A      BDOS multi sector count for read/write
4B      BDOS error mode:
          FF = do not display error messages, return to current program
          FE = display error messages, return to current program
          else terminate current program on error and display message
4C - 4F Drive search chain: up to 4 drives can be specified
        0 = current default drive, 1 = A:, 2 = B:, etc.
        If program or file is not found on specified drive, the search
        chain will be used and each drive in the list will be tried
        in sequence until it is found.
50      Temporary file drive: 0 = default, 1 = A:, etc
51      Error drive: number of drive where last I/O error was encountered
        (0 = default, 1 = A:, etc)
52 - 53 reserved for system use
54      BIOS flag to indicate disk changed
55 - 56 reserved for system use
57      BDOS flags: bit 7 set, system displays expanded error messages
        (default is set)
58 - 59 date in days in binary since 1 jan 78
5A      hour in BCD
5B      minutes in BCD
5C      seconds in BCD
5D - 5E start of common memory (E000)
5F - 61 reserved for system use
62 - 63 top of user TPA  (from vector at 0006 - 0007: entry point to BDOS)
```

BIOS Jump table:

(BIOSBASE= F000 in DEC and MAY versions
F400 in AUG version)

(NOTE: the first byte of each group is a Z80 jump instruction to the address contained in the next two bytes.)

| Address | Function | BIOS Function number |
|---|---|---|
| BIOSBASE+00 to +02 | cold boot | 0 |
| BIOSBASE+03 to +05 | warm boot | 1 |
| BIOSBASE+06 to +08 | check CONSOLE input status | 2 |
| BIOSBASE+09 to +0b | read CONSOLE character | 3 |
| BIOSBASE+0c to +0e | write CONSOLE character | 4 |
| BIOSBASE+0f to +11 | write LIST character | 5 |
| BIOSBASE+12 to +14 | write AUXILIARY OUT character | 6 |
| BIOSBASE+15 to +17 | read AUXILIARY INPUT character | 7 |
| BIOSBASE+18 to +1a | move to track 0 on selected disk | 8 |
| BIOSBASE+1b to +1d | select disk drive | 9 |
| BIOSBASE+1e to +20 | set track number | 10 |
| BIOSBASE+21 to +23 | set sector number | 11 |
| BIOSBASE+24 to +26 | set DMA address | 12 |
| BIOSBASE+27 to +29 | read specified sector | 13 |
| BIOSBASE+2a to +2c | write specified sector | 14 |
| BIOSBASE+2d to +2f | check LIST status | 15 |

| | | |
|---|---|---|
| BIOSBASE+30 to +32 | translate logical to physical sector | 16 |
| BIOSBASE+33 to +35 | check CONSOLE output status | 17 |
| BIOSBASE+36 to +38 | check AUXILIARY INPUT status | 18 |
| BIOSBASE+39 to +3b | check AUXILIARY OUTPUT status | 19 |
| BIOSBASE+3c to +3e | get address of character I/O table | 20 |
| BIOSBASE+3f to +41 | initialize character I/O devices | 21 |
| BIOSBASE+42 to +44 | get address of disk drive table | 22 |
| BIOSBASE+45 to +47 | set # of logical sectors to read/write | 23 |
| BIOSBASE+48 to +4a | force I/O buffer flush | 24 |
| BIOSBASE+4b to +4d | memory move | 25 |
| BIOSBASE+4e to +50 | get or set time | 26 |
| BIOSBASE+51 to +53 | select memory bank | 27 |
| BIOSBASE+54 to +56 | specify bank for DMA operation | 28 |
| BIOSBASE+57 to +59 | set buffer bank | 29 |
| BIOSBASE+5a to +5c | call user system functions | 30 |

```
BIOSBASE+d3 to +d4  pointer to physical device table
                    (points to $f7fd in AUG and $f3e1 in DEC and MAY)

DEVTBL = Physical device table, each entry is 8 bytes long:

 device name (6 bytes, padded with spaces)
 mode byte   (1 byte)
 baud rate   (1 byte)
```

| Mode Byte Component | Device Function |
|---|---|
| 0000 0001 | device can do input |
| 0000 0010 | device can do output |
| 0000 0011 | device can do both |
| 0000 0100 | supports software selected baud rate |
| 0000 1000 | supports serial I/O |
| 0001 0000 | supports XON/XOFF |

| Baud Rate Byte | Function | Baud Rate Byte | Function | Note: |
|---|---|---|---|---|
| 0 | none | 9 | 1800 | For all practical uses, C128 |
| 1 | 50 baud | a | 2400 | CP/M will not support rates |
| 2 | 75 | b | 3600 | over 1200 baud because of |
| 3 | 110 | c | 4800 | software overhead. |
| 4 | 135 | d | 7200 | |
| 5 | 150 | e | 9600 | |
| 6 | 300 | f | 19200 | |
| 7 | 600 | | | |
| 8 | 1200 | | | |

```
C128 devices are: KEYS, 80COL, 40COL, PRT1, PRT2, 6551, and RS232

BIOSBASE+d7 to d8   pointer to drive table (List of DPH addresses).
                    (points to FBD1 on DEC and MAY versions and FB78 on AUG)

FC00 to FD00  INTBLOCK   ; contains all FD's.  This is the pointer to the
    interrupt vector (FDFD). A hardware quirk of the C128 requires from FC00
    to FD00 to be all FD's else the interrupt pointer address will not be
    correctly specified on the Z-80 address bus. If you use this seemingly
    unused area for your own programs, it will crash on a random basis if an
    interrupt occurs.  BEWARE!!

PARMBLOCK     ; BIOS parameter working storage for passing to 8502
FD01 VICCMD   ; BIOS 8502 command to execute
FD02 VICDRV   ; device number to execute on printer.
                for disk I/O:

              Bit Value   Drive
              -------------------------
              0000 0001   unit 8, drive 0   (default drive A: and E:)
              0000 0010   unit 9, drive 0   (default drive B:)
              0000 0100   unit 10, drive 0  (default drive C:)
              0000 1000   unit 11, drive 0  (default drive D:)
              1000 0001   unit 8, drive 1
              1000 0010   unit 9, drive 1
              1000 0100   unit 10, drive 1
              1000 1000   unit 11, drive 1

FD03 VICTRK    ; track number to execute disk operations on,
                 or secondary address for printer operations
FD04 VICSECT   ; sector number for disk operations.  For MFM
                 disks side 1, use $80 + sector #
FD05 VICCOUNT  ; number of sectors to read/write for disk I/O,
                 number of characters to print on printer or low
                 byte of address to jump to for custom 8502 code.
FD06 VICDATA   ; various data/status items for disk and printer or hi
                 byte of address to jump to for custom 8502 code.
FD07 CURDRV    ; current logical drive using device A:  (0 = drive A:
                                                         4 = drive E:)
FD08 FAST      ; drive type flags (updated after each drive I/O
                 operation), bit on=drive is fast (1571 or 1581)
                                off=drive is slow (1541 type)

                 Bit      Drive
                 ------------------------
                 0        A:  (device 8)
                 1        B:          9
                 2        C:         10
                 3        D:         11
                 4-7      (not currently used)

FD09 KEYTBL       ; pointer to key scan definition table (1290)
FD0B FUNTBL       ; pointer to function key table (1000)
FD0D COLORTBLPTR  ; pointer to logical colour table (13F0)
FD0F FUNOFFSET    ; offset to current function key (0 if none)

FD10 SOUND1       ; pointers for various clicks, beeps etc.
FD12 SOUND2
FD14 SOUND3
```

The following storage locations are used by the general BIOS and BDOS function calls. Values for BIOS 8502 storage (FD01 etc above) are taken from here:

```
FD16 @TRK       ; track number used in BDOS and BIOS routines
FD18 @DMA       ; address of disk data buffer
FD1A @SECT      ; sector number used in BDOS and BIOS routines
FD1C @CNT       ; numberof sectors to read/write
FD1D @CBNK      ; current bank   (0 or 1)
FD1E @DBNK      ; bank for disk data buffer  (0 or 1)
FD1F @ADRV      ; absolute drive code  (0 = A:, 1 = B:, etc)
FD20 @RDRV      ; relative drive code (same as for VICDRV
                  outlined above)

FD21 CCPCOUNT   ; # 128 byte records in CCP.COM file.
```

```
FD22 STATENABLE   ; general purpose status flags:

     Bit     Meaning (when set to 1)
     ---------------------------------------------------
     7       allow 8-bit key codes
             (key values above $80 will not result in
             special functions defined in key scan table)
     6       track cursor on 40-col screen
             when doing input on 80-col screen
     5 to 1  (not used?)
     0       display disk status on bottom line of screen

FD23 EMULATIONADR  ; pointer to screen emulation code address (used to
     switch between VT100 and ADM31 modes. Can be pointed to custom code
     (in bank 0) for other types of emulation.)

FD25 USARTADR      ; pointer to external 6551 address (not normally used)

FD27 INTHL         ; temporary storage for CPU registers during
FD3D INTSTACK      ; interrupts, BDOS and BIOS calls, etc.
FD3D USERHLTEMP
FD3F HLTEMP
FD41 DETEMP
FD43 ATEMP

FD44 SOURCEBNK     ; source bank for inter-bank memory move   (0 or 1)
FD45 DESTBNK       ; destination bank for inter-bank move     (0 or 1)

FD46 MFMTBLPTR     ; pointer to MFM disk parameter table (DPT)

FD48 PRTCONV1      ; pointer to character conversion routine for PRT1
                     printer (normally ASCII to PETSCII)
FD4A PRTCONV2      ; pointer to character conversion routine for PRT2

FD4C KEYFXFUNCTION ; pointer to dispatch table for executing extended
     key codes functions F0 to FF. This table can be patched to add your
     memory resident special functions.

FD4E XXDCONFIG     ; RS-232 configuration register
FD4F RS232STATUS   ; RS-232 status register

  Bit   Meaning                       | Bit   Meaning
  ------------------------------------|------------------------------
   7    0 = ready to receive data     |  3    1 = framing error
   6    0 = idle   1 = busy           |  2    1 = other error
   5    1 = data in buffer que        |  1    1 = receiving data
   4    1 = parity error              |  0    1 = ready to send data
  ----------------------------------------------------------------------

FD50 XMITDATA      ; RS-232 sending data buffer
FD73 RXDBUFCOUNT   ; number of characters in RS-232 buffer
FD74 RXDBUFPUT     ; temporary storage for RS-232 variables
FD75 RXDBUFGET
FD76 RXDBUFFER     ; RS-232 data receiving buffer (60 characters)

FDFD INTVECTOR     ; main entry point to interrupt routine (jump to
                     routine elsewhere)

FE00 @BUFFER       ; 256-byte disk and general I/O buffer (you can put
               your own code here if it is OK to overwrite when not in use)

FF00 FORCEMAP      ; MMU configuration register
FF01 BANK0         ; force to MMU value of 3F
FF02 BANK1         ;                       7F
FF03 IO            ;                       3E
FF03 IO0           ;                       3E
FF04 IO1           ;                       7E

FFD0 ENABLEZ80     ; 8502 code to switch to Z-80 mode
FFDC RETURNZ80

FFE0 ENABLE6502    ; Z-80 code to switch to 8502 mode
FFEE RETURN6502

*************************************************************
```

```
BANK 1:
-------

BANK 1 is the transient program bank. All transient programs operate in
this bank. The MMU configuration value is $7F.

0000-0002  Jump to BIOS warm start entry BIOSBASE + 3

0004       high nibble = current user number
            low nibble = current default drive (0 to f, 0=A, etc.)

0005-0007  Jump to BDOS entry point (address stored in 0006-0007)
           NOTE: This address also marks the end of the TPA. It can be
           artificially lowered for use by resident programs.

0008-004f  reserved for RST 1 to 7 (Z-80 restart instructions)

(NOTE: Locations 0050 to 007b are set automatically by the CCP when
       a transient program is loaded. The transient program can
       then check this areas for parameters which may have been
       passed from the console in the form of a command tail.)

0050       drive from which latest transient program was loaded,
           1=A, 2=B, ... 16=P

0051-0052  pointer to password of first operand in command tail (points to
           a location in the CCP buffer starting at 0080). It is set to 0
           if no password specified.

0053       length of first password. Set to 0 if no password.

0054-0055  pointer to password of second operand in command tail. Set to
           0 if no password specified.

0056       length of second password. Set to 0 if no password.

005c-007b  default parsed file control block (FCB) area:
           005c-006b  initialized from first command tail operand
           006c-007b  initialized from second command tail operand

     (NOTE: The FCB areas are normally 32 bytes long each. Thus, the second
     FCB area must be moved to an unused area before the first FCB area can
     be used or else it will be overwritten.)

007c       current record position for default FCB 1.

007d-007f  current random record position for default FCB 1.

0080-00ff  default 128 byte disk buffer and CCP input buffer.

0100-e9ff  58K transient program area (TPA) DEC and MAY versions
    -edff  59K TPA on AUG version

e000-ffff  common with BANK 0 (top of TPA, also BDOS, BIOS, etc.)

***************************************************************************

Bank 2 : MMU value = $3E

This bank, which I have arbitrarily called BANK 2, is mostly the same as
Bank 0 with the following exception:

1000 to 13FF    VICCOLOR  ; colour map for 40-col screen

This bank must also be in context to access the MMU chip registers at D500
in the Z-80 I/O mapped area.

***************************************************************************
```

Z-80 I/O mapped area:
--------------------

In addition to the normal memory mapping which includes RAM and ROM, the Z-80 has another addressing mode which is called I/O mapping. This is used to access the chip registers and is similar to memory mapping except the Z-80 IN and OUT instructions must be used instead of LD type instructions. The two most commonly used ones are:

```
IN A,(C)   which will read the value of I/O addressed by .BC into the
           .A register

OUT (C),A  which will write the value of .A to the I/O addressed by .BC
```

In both cases, .BC is a 16-bit address and .A is an 8-bit value.

The C128 I/O chips appear at their normal address locations in the I/O area as outlined below and can be programmed directly by the experienced user. Note that you must be careful when playing with register values because CP/M expects most of them to be set in certain ways. Changing these settings may cause a system crash. The remainder of the address space is taken up by "bleed through" from the underlying bank 0 RAM. An exception to this is the space from 0000 to 0FFF which actually contains bank 0 RAM from address D000 to DFFF.

```
D000 - VIC                     DE00 external 6551 USART
D400 - SID                     DE00 TXD6551 (send & receive data register)
D500 - MMU                     DE01 RESET6551 (write only)
D600 - 8563 80-column chip     DE01 STATUS6551 (when read)
D800 - VICCH 40-col colour map DE02 COMMAND6551
DC00 - CIA #1                  DE03 CONTROL6551
DD00 - CIA #2                  DF00 RAM expander DMA controller chip
```
******************************************************************************

8502 Mode
---------

Surprisingly, the CP/M side of the C128 also makes use of the 8502 side, including the KERNAL ROMs for some of the low-level I/O operations. In 8502 mode, RAM from 0000 to 01FF is common between the banks. Note that this is different than normal C128 mode which has common RAM from 0000 to 03FF. The most important implication of this that the KERNAL JSFAR, JMPFAR, INDFET, INDSTA, and INDCMP routines for bank manipulation cannot be used because they all rely on code which is no longer in common RAM (it is above 0200).

Most of the memory map is similar to the Z-80 side with the exception of the low end. This low end area is very similar to that used in normal C128 mode except as noted below.

Bank 0 (MMU value 3F):
---------------------

```
             ; BIOS 8502 working storage
000A         ; current printer device number
000B         ;          printer secondary address
000C         ;          disk drive data channel #
000D         ;          disk drive command channel #
000E         ;          disk drive device #
000F to 0012 ; temporary storage pointers

0013 to 008F ; unused zp RAM where you can stash your own 8502 code
             ; (pointers normally used in C128 mode, but not required
             ;  in CP/M mode)

0090 to 00CA ; KERNAL pointers required for disk and printer I/O
             ; same as C128 mode

00CC to 00FF ; unused zp RAM

0100 to 0200 ; various KERNAL pointers, 8502 stack, etc.  Some
               unused locations but too chopped up to put code here.

0201 to 02FF ; basically unused, you can put some code here if you wish

0300 to 0333 ; system vectors

0334 to 0361 ; unused?

0362 to 037F ; logical file tables

0380 to 09FF ; unused?

0A03         ; FF=50 Hz, PAL; 0=60 Hz, NTSC

0A1C         ; serial bus fast flag
```

Other areas up to 0FFF are used sporadically. Large open spaces in table buffer (0B00 to 0BFF) and C128 mode RS-232 buffers (0C00 to 0DFF).

```
1000 and up  ; common with Z-80 side
```

The following area is used when the 8502 is switched in during CP/M operations:

```
2600-2a40  BIOS8502 ; 8502 BIOS code (all code here is in standard 8502 ML)
```

| | BIOS8502 function | Jump Table Entry AUG | Jump Table Entry MAY/DEC | Points to code at AUG | Points to code at MAY/DEC |
|---|---|---|---|---|---|
| -1 | reset | 260f | 2614 | 2625 | 262e |
| 0 | initialize | 2611 | 2616 | 2654 | 265d |
| 1 | read 1541 | 2613 | 2618 | 2682 | 2690 |
| 2 | write 1541 | 2615 | 261a | 26b0 | 26be |
| 3 | read 1571/81 | 2617 | 261c | 26f9 | 2707 |
| 4 | wrt 1571/81 | 2619 | 261e | 26f6 | 2704 |
| 5 | inquire disk | 261b | 2620 | 270e | 271c |
| 6 | query disk | 261d | 2622 | 2740 | 274e |
| 7 | printer | 261f | 2624 | 2776 | 2784 |
| 8 | format dsk | 2621 | 2626 | 27ac | 27e2 |
| 9 | user code | 2623 | 2628 | 262b* | 2634 |
| a | 1750 read | n/a | 262a | n/a | 2820 |
| b | 1750 write | n/a | 262c | n/a | 2823 |

* NOTE: There is an error in the code at $262b of the AUG version. The byte value is $c3 and it should be $6c for a JMP (xxxx). This means that 8502 BIOS subfunction 9 (jump to custom user 8502 code) does not work on the AUG version unless you first correct this bug!!

BIOS8502 IRQ, BRK and NMI vectors point to 29ae on the AUG version and 29f0 on the MAY and DEC ver.

Other I/O vectors are unchanged.

Bank 1 (MMU value 7F):
---------------------

```
0000 to 01FF   ; common with BANK 0

0200 to FFFF   ; common with Z-80 memory map in BANK 1
```

# WHEREIS

## *Notes from the CP/M Plus workbench*

**by Adam Herst**
*Copyright © 1988, Adam Herst*

It is not easy to find a specific file in an unknown user area on a disk in an unknown drive. C-128 CP/M disks can hold up to 128 files. These files can be isolated across 16 user areas. Up to 16 drives can theoretically be attached to a CP/M system. The only tool provided with CP/M Plus with which to find a specific file or files across all drives and user areas is the transient version of the DIR command.

You can search for a file across user areas and across disk drives with the command:

DIR d:filename.typ{,d:filename.typ} [USER=ALL]

where d is the drive to search, filename.typ is the name of the file for which to search, and the braces ({ }) indicate an optionally repeated argument.

There are a number of drawbacks to this method. The transient version of DIR is a large file and can be slow to load. It is a generalized tool making it slow to execute. It displays extraneous, as well as desired, information. The drives to be searched must be explicitly stated on the command line. The information returned by DIR concerning file location is not in a format suitable for use in batch processing. Finally, even though the transient version of DIR is exclusively a CP/M 3.0 command, it does not manipulate the program return code.

WHEREIS addresses all of these problems. Written in 8080 assembly language and assembled with MAC and HEXCOM (supplied as part of the DRI special offer package), WHEREIS is small, fast and specialized for this task. If no drive is specified for the search, all drives in the drive search path are searched. Finally, the program return code will be set to *unsuccessful* if no match to the file specified on the command line is found.

WHEREIS prints the drive code, user area code, and filename.typ for the found file(s) in the form:

duu:filename.typ

where 'd' is the drive the file was found on, 'uu' is the user area it was found in, and 'filename.typ' is the filename and filetype of the found file.

WHEREIS is invoked with a command of the form:

WHEREIS d:filename.typ

where 'd:' is an optional valid drive specification, 'a:' to 'p:', and 'filespec' is a valid filename and filetype with appropriate wildcards. If an ambiguous filespec is given, all matches will be returned. If no drive specification is given, all drives in the drive search path will be searched.

The source code is well commented and the program is (I hope) self explanatory. A few points are worth explicit note.

WHEREIS uses two system services that are available only under CP/M Plus: get/set SCB, and get/set program return code. WHEREIS uses the get/set SCB call to determine the drive search path, and uses the get/set program return code to set the program return code to *unsuccessful* if no file match is found. (An incomplete description of the SCB data structure and the interpretation of return codes is provided in the documentation of the appropriate system call in the *CP/M Plus Programmers System Guide* in the volume supplied with the DRI special offer package.)

Making these calls under CP/M 2.2 will yield unpredictable results. WHEREIS checks the version number of the operating system as a courtesy to CP/M 2.2 users. A comparison is made for a returned version code of 0h indicating that the version of the operating system is pre-3.0. (Multiple versions of CP/M 3.0 are available. CP/M on the C-128 returns a version number of 3.1. An explicit check for CP/M 3.x would require 16 comparisons). If the version of CP/M is pre-CP/M 3.0, WHEREIS will abort with a reminder that CP/M 3.0 is required.

Both of the system services used by WHEREIS that are exclusive to CP/M 3.0 access the System Control Block (SCB), either directly or indirectly. The SCB is a 100-byte data structure containing system flags and variables that can be accessed by the assorted modules that comprise the CP/M operating system. Some SCB flags and variables can be queried or set through CP/M system calls. Others must be queried or set through system call 31h and a user-defined data structure called the SCB Parameter Block (SCBPB).

One variable that can be manipulated directly is the program return code. The program return code is a one-word variable contained in the SCB. It can be used to pass a value to a chained program and is tested by the CCP during the execution of submit files to conditionally execute command lines. (See the article "Exploring SUBMIT" for more information on conditional command lines.) In a submit file, a program return code between 0000h and feffh inclusive indicates successful program termination, while those between ff00 and fffeh inclusive indicate an unsuccessful program termination.

The program return code can take on the following values:

```
0000h - feffh   successful return
ff00h - fffeh   unsuccessful return
0000h           CCR initialization value
ff80h - fffch   reserved for system use
fffdh           fatal BDOS error
fffeh           CTRL-C
```

The codes between ff00h and ff7fh are available to the programmer to indicate an unsuccessful termination. WHEREIS sets the program return code to ff01h, unsuccessful, on execution. The return code is set to 0000h, successful, when a file match is found.

CP/M 3.0 provides for the definition of a drive search path. Up to four drives can be included in the CCP command search. The drive search path is also stored in the SCB.

At the command level, the drive search path can be queried or set with the DEVICE command. Issued without options, DEVICE will display the current drive search path along with other environment information. The drive search path can be set with the command:

DEVICE d{,d}

where 'd' is the letter of a valid drive, 'a' to 'p', or an asterisk, '*', to indicate the default drive. Up to four drives can be specified in the drive search path separated by commas.

At the system level, there is no call to directly query or set the drive search path. The drive search path must be queried or set using the SCB system call and the associated SCBPB. The SCBPB consists of three fields: the offset, the get/set code, and the set value. The offset is a single byte indicating the number of bytes from SCB base of the field to access. The get/set code is a single byte that indicates whether the call will get or set the indicated SCB field. A code of 00h indicates a get operation, a code of 0ffh that a byte should be set, and a code of 0feh that a word should be set. The set value is a byte or word containing the value that the indicated SCB field should be set to.

The drive search path occupies four bytes in the SCB, starting at the byte with an offset of 4ch and ending at the byte with an offset of 4fh. Each byte in the drive path chain represents a disk drive with codes 01h-0fh representing drives A: to P: respectively. A drive code of 00h represents the default drive. A code of 00ffh indicates that the following bytes in the drive path chain have not been set, i.e. this is the end of the chain.

WHEREIS accesses the drive search chain through a function with two entry points, PATHF and PATHN. Successive calls to the function leave the code of the next drive in the drive search path or a 00h (indicating the end of the path) in the accumulator. Drive codes are retrieved from the drive search path until a code of 00ffh is retrieved or until the SCB offset points past the four-byte drive path chain. (This latter indicator is used to avoid searching the drive path chain when a drive is specified on the command line for searching. The offset is immediately set past the drive path chain to end the search after the specified drive.) A call to PATHF initializes the SCBPB and other flags and returns the code of the first drive in the path in the accumulator. Subsequent calls to PATHN will return subsequent drive codes.

The bulk of the function ensures that the code of the default drive is not returned twice if it is included in the path through both an implicit (the default drive code) and an explicit (the drive code) reference. The default drive can be specified as part of the drive search path. If the default drive also is named in the drive search path, it will be searched twice - once when its explicit drive code is retrieved from the chain and once when the default drive code is retrieved from the chain. To avoid this situation a flag, DRVFLG, is maintained. The low nybble contains the drive code of the default drive. The high nybble contains the 'default drive already returned' flag - it is set to 0 if the default drive has not been returned, and to 0fh if it has been returned. If the default drive has been returned, the drive code of the next drive in the path will be returned instead.

WHEREIS is very useful in locating the files that inevitably get spread across user areas and disk drives. Another use is to allow simple conditional command execution in submit files. The submit file I use to drive MAC and HEXCOM uses WHEREIS to check for the existence of the appropriate .HEX file before invoking HEXCOM on a conditional command line (the first character in the line is a colon followed by a space). If the file is not found, WHEREIS sets the program return code to *unsuccessful* and no time is wasted invoking HEXCOM on a non-existent file. There is probably a place for WHEREIS in your toolkit.

**Listing: whereis.asm**

```
; whereis (c) 1988 adam herst
;
;
;       where is the match to the file specification on command line
;       expand wild cards
;       search drive if specified or search all drives in drive path
;       search all user areas
;       return the drive specification in the form DUU:filename.typ
;       set program return code to unsuccessful if no match found
;
```

```
; BDOS services
        ConOut  equ     2h          ; print character
        PrStr   equ     9h          ; print string
        VerNum  equ     0ch         ; return cpm version number
        SrchF   equ     11h         ; search for first file name
        SrchN   equ     12h         ; search for next file name
        CurDrv  equ     19h         ; get current drive code
        SetDma  equ     1ah         ; set dma buffer
        GetUsr  equ     20h         ; get/set user number
        GetScb  equ     31h         ; get/set scb
        RCode   equ     06ch        ; get/set program return code
; system pointers
        FCB     equ     5ch         ; FCB buffer pointer
        DMA     equ     400h        ; DMA buffer pointer
        BDOS    equ     5h          ; BDOS function pointer
; symbols
        CR      equ     0dh         ; ascii carriage return
        LF      equ     0ah         ; ascii line feed
        RCODE0  equ     0000h       ; successful return code
        RCODE1  equ     0ff01h      ; user unsuccessful return code

        org     100h

; exit with message if not cpm 3.0
        mvi     c,VerNum
        call    BDOS
        ora     a
        jnz     ISCPM3
        mvi     c,PrStr
        lxi     d,ISCPM2
        call    BDOS
        ret
ISCPM2  db      'whereis requires CP/M 3.0',CR,LF,'$'

; set dma buffer
ISCPM3  mvi     c,SetDma
        lxi     d,DMA
        call    BDOS
; set return code to unsuccessful
        mvi     c,RCODE
        lxi     d,RCODE1
        call    BDOS
; check if a drive is specified for search
        lxi     h,FCB       ; point to drive code in fcb
        mov     a,m         ; get drive code from fcb
        ora     a           ; is it the default drive code?
        jz      DRIVE       ; yes so start searching drive path
        lxi     h,DRVOFF    ; point to offset for drive path in subroutine
        mvi     m,04fh      ; set offset to end of drive path to force exit
        jmp     USER        ; search user areas
; search drives in drive path
DRIVE   call    PATHF       ; get first drive in path, leave in a
NODRV   cpi     0h          ; is there a drive in the path?
        rz                  ; no so exit
        lxi     h,Fcb       ; point to drive code in fcb
        mov     m,a         ; put drive code in fcb
; search user areas
USER    mvi     c,GetUsr    ; start at user area 0
        mvi     e,0h
        call    BDOS
; search for first match to the file name in the FCB
FILE    mvi     c,SrchF
        lxi     d,FCB
        call    BDOS
; check return status
        cpi     0ffh        ; found file?
        jz      NXTUSR      ; no so set next user number
        call    PFSPEC      ; yes so print filespec
; search for next match in this user area
NXTFIL  mvi     c,SrchN
        lxi     d,FCB
        call    BDOS
; check return status
        cpi     0ffh        ; found file?
        jz      NXTUSR      ; no so set next user number
        call    PFSPEC      ; yes so print filespec
; do next file
        jmp     NXTFIL
; do next user area
NXTUSR  mvi     c,GetUsr
        mvi     e,0ffh      ; get user number from system
        call    BDOS
        inr     a           ; increment user number
        cpi     10h         ; is it user number 16?
        jz      NXTDRV      ; yes so do next drive in path
        mvi     c,GetUsr
        mov     e,a         ; set new user number
        call    BDOS
        jmp     FILE        ; search the next user area for  first file
; do next drive in path
NXTDRV  call    PATHN       ; get next drive code, leave in a
        jmp     NODRV       ; search the drive if there is one
; create and print the filespec of dma entry in a
PFSPEC
; do user number
        lxi     h,DMA       ; point to user number in first entry
        mvi     b,0h        ; entry counter to 0
ENTRY   cmp     b           ; is it the right dma entry?
        jz      NUMBER      ; yes so put user number in filespec buffer
        lxi     d,20h       ; point to user number next entry
        dad     d
        inr     b           ; increment entry counter
        jmp     ENTRY       ; check if the right entry
NUMBER  mov     a,m         ; get user number from dma entry
        cpi     10d         ; is it less than 10
        jnc     ONE         ; yes so 1 as first digit in user number
        mvi     a,'0'       ; 0 as first digit in user number
        sta     FSPEC+1
        jmp     SECOND      ; do second digit in user number
ONE     mvi     a,'1'       ; put a 1 in filespec buffer
        sta     FSPEC+1
SECOND  mov     a,m         ; get user number from file entry in DMA buffer
        cpi     10d         ; is it less than 10
        jc      NOSUB       ; no so don't subtract
        sui     10          ; subtract the ten
NOSUB   adi     30h         ; make it printable
        sta     FSPEC+2     ; put it in the filespec buffer
; do filename
; copy file name from dma buffer to filespec buffer
        inx     h           ; point to filename in current entry
        lxi     d,FSPEC+4h  ; point to filename in filespec buffer
        mvi     b,8h        ; set counter for filename length
FNAME   mov     a,m
        stax    d
        inx     h
        inx     d
        dcr     b
        jnz     FNAME
        mvi     b,3h        ; set counter for filetype length
        inx     d           ; point to filetype in filespec buffer
FTYPE   mov     a,m
        stax    d
        inx     h
        inx     d
        dCR     b
        jnz     FTYPE
; do drive letter
        lxi     h,FCB       ; point to drive code in fcb
        mov     a,m         ; get the drive code
        adi     40h         ; make it printable
        lxi     h,FSPEC     ; point to drive code in filespec buffer
        mov     m,a         ; put drive code in filespec buffer
; print the filespec
        mvi     c,PrStr
        lxi     d,FSPEC
        call    BDOS
```

```
; set return code to successful
        mvi     c,RCode
        lxi     d,RCODE0
        call    BDOS
; return from PFSPEC
        ret
; return the next drive in the drive path in a
;
; initialize for first drive in path
PATHF   mvi     c,CurDrv    ; get current drive code
        call    BDOS
        inr     a           ; add drive code offset
        lxi     h,DRVFLG    ; point to drive flag
        mov     m,a         ; set flag, low bits to drive code, high bits to 0
        lxi     h,DRVOFF    ; point to offset of current drive in path
        mvi     m,4bh       ; set offset to first drive in path - 1
; get next drive in path
PATHN   lxi     h,DRVOFF    ; point to scb offset value for previous drive
        mov     a,m         ; get the offset
        inr     a           ; set offset for current drive
        mov     m,a         ; put offset value of current drive
        cpi     50h         ; is current offset pointing past last drive in path?
        jz      RETEND      ; yes so return
        lxi     h,SCBPB     ; point to offset in scb parameter block
        mov     m,a         ; put offset of current drive in scbpb offset
        inx     h           ; point to get/set code in scbpb
        mvi     m,0h        ; set for get operation
        mvi     c,GetScb    ; get the current drive in the drive path
        lxi     d,SCBPB     ; point to scb parameter block
        call    BDOS
        cpi     0ffh        ; is it the end of the path?
        jz      RETEND      ; yes so return no more drives code
        mov     b,a         ; store current drive code from path
        cpi     0h          ; is it the default drive code?
        jz      CHKFLG      ; yes so check if default drive already returned
        lxi     d,drvflg    ;  point to the drive flag
        ldax    d           ; get drive flag
        ani     0fh         ; mask off default drive returned bits
        cmp     b           ; is it the drive code of the default drive?
        jnz     RETDRV      ; no so return drive code
; check if default drive has already been returned
CHKFLG  lxi     d,DRVFLG        ; point to drive flag
        ldax    d           ; get drive flag
        ani     0f0h        ; mask off drive code
        cpi     0f0h        ; has it already been searched?
        jz      PATHN       ; yes so get next drive in path
        ldax    d           ; get drive flag, searched bits are already 0
        mov     b,a         ; store drive code of current drive
        ori     0f0h        ; mask on default returned flag to returned (0fh)
        stax    d           ; put default returned flag in drive flag
; return drive code of current drive in a
RETDRV  mov     a,b         ; leave the drive code in a
        ret                 ;
RETEND  mvi     a,0h        ; leave no more drives code in a
        ret
; EXIT
EXIT    ret
; filespec buffer
FSPEC   db      '   :        .   ',CR,LF,'$'
; 'where' version number
WHERE   db      'v1.2$'
; scb parameter block
SCBPB
SCBOFF  db      ' '         ; offset byte, set to start of drive path - 1
SCBCOD  db      ' '         ; get/set code
SCBVAL  dw      '  '        ; byte or word value for set operations
; drive flag
DRVFLG  db      ' '         ; high nybble = 0h, default not searched
                            ;             = Fh, searched
                            ; low nybble = default drive drive code
; scb offset for current drive in drive path
DRVOFF  db      ' '

        end
```

# I Do Windows (on the C128)

## *A program for paneless print positioning*

**by Jim Butterfield**


The usual way to set up a screen window on the C128 is to use the command WINDOW. It's easy and natural, and you can set up the size and other details on the spot.

But WINDOW has its limitations. It seems to be devised for a one-time setup. When you enter the window, you'll be at its top; great for clearing the area and starting a new display. But it's not too handy if you want to add new information to the bottom of an existing window.

I suppose you could get around this with the Escape sequence, CHR$(27)+"W". This would scroll the window *down*, leaving space on the top line for the new information. So, PRINT CHR$(27)+"W"+"NEWSTUFF" would print NEWSTUFF on a fresh line at the top of the window. But it seems unnatural for the screen to scroll that way.

There's another approach to doing windows. It involves using the ESC-T ("top") and ESC-B ("bottom") sequences. Here's the idea: if you place the cursor at the desired top/left positioning of the window, and then print ESC-T, that part will be set... then you can move the cursor to the bottom/right and print ESC-B to fix the rest of the window. And as a byproduct of this method, you'll be positioned at the bottom of the window... a new line of data will scroll previous material upward.

This method also works on the Plus/4 and Commodore 16, which don't have a WINDOW command. They do, however, recognize the ESC codes, as does the B-128.

Program "window128" demonstrates this way of doing things. A series of numbers (your choice) are generated. If the number is prime, it will be put into the right-hand window. Otherwise, it is tested for division by two, three, five, and seven, and put into the appropriate window depending upon the smallest divisor. If the number is not prime, but has a divisor higher than seven, it's put into a window labelled HIGH.

Here's how the program sets up the various windows. It first places the cursor at the appropriate top/left position, and then prints W$. Note that W$ is defined in line 270; it contains all the ingredients for making a window.

The first part of W$, line 270, is CHR$(27)+"T". That fixes the top/left part of the screen at wherever the cursor is located. Next, W$ contains cursor movements... a number of cursor-rights followed by some cursor-downs. The cursor will now be moved a fixed distance to the desired bottom-right position... at which time, W$ fixes this point with a CHR$(27)+"B". The window is sized, and the cursor is at the bottom.

The program starts all windows at the same row, but positions the window horizontally with a TAB command in line 610. Depending on the value of variable K (1 to 6), the window will be set up at its proper location across the screen. Note that right after the TAB, we print W$ and bring the window into existence.

Windows are exited by printing two successive "HOME" characters. You'll see this in line 600, which is where we start to set up a new window, and line 640, where the program ends. (We really wouldn't want "READY." to print within our window).

About the factor calculation: The loop from line 300 to 630 walks through the range of numbers the user has requested. Array F() contains a list of factors: two, three, five, and seven, and the loop from 320 to 350 tries them all.

Before we do this trial division, however, we do a special check for the 'root factor.' For example, two divides by two... but we don't want to put it in the 'twos' column, because it's prime. So line 310 tests for these low numbers.

If we don't find a factor of seven or lower, we must try trial division. The smallest possible factor is now eleven, and the highest is the square root of the number in question. We need to try odd numbers only... and this we do in the loop from 390 to 420.

The program is set up to use a 40-column screen. If you have 80 columns, you might like to try modifying the program to allow more columns... or to make the existing columns bigger.

*The listing "window128" appears on page 53. -Ed.*

# The Edge Connection

## *Comparing three C128 assemblers*

**by Joel M. Rubin**

This is a comparison of three assemblers available for the C128: LADS (COMPUTE! U.S. $16.95 if typed in, $29.90 with disk, COMPUTE! Publications, P.O. Box 5038, F.D.R. Station, New York, NY 10150); Buddy (Pro-Line, approximately $30 in U.S. stores under the name "Power" from Better Working/Spinnaker, Spinnaker Software Corp., One Kendall Sq., Cambridge, MA 02139); and Merlin 128 (U.S. $69.95 direct by mail, less in stores), Roger Wagner Publishing, Inc., 10761 Woodside Ave, Suite E, Santee, CA 92071).

LADS is in a COMPUTE! book, "128 Machine Language for Beginners", by Richard Mansfield. The book contains both the assembly language source code and a hex dump, with COMPUTE!'s "MLX" correction code, every eight bytes. (The hex entry program, MLX, is included in the book, and there is also COMPUTE!'s BASIC proofreader, similar to Verifizer, to help you type in BASIC programs, such as MLX.)

*Seeing how a full-blown assembler is written is, in itself, a good lesson in assembly language...*

LADS source code is written in the familiar environment of the BASIC editor. Thus, you can use any BASIC programming tools which you may have, such as a search-and-replace utility. Since LADS lives relatively low in memory, at $2710, the length of source code may be limited, although LADS allows linked source code files. LADS has relatively few pseudo-ops. For example, you can set up a byte or a string, but not a word (at least not very conveniently).

Also, even when you save the object code to disk, LADS pokes it to memory, at the location you specify. In one mode, you can use RAM 1, between $400 and $ffef, but this might still make things difficult if, for example, you were writing disk buffer code at $300.

The big advantage of LADS is that since you get the source code, you can - if you are skilful enough - add features or modify the code to enhance LADS as you see fit. Also, seeing how a full-blown assembler is written is, in itself, a very good lesson in assembly language programming. The book is probably worth having, in and of itself, as a tutorial on 6502 assembly language.

**The power of Buddy**

Buddy, or Power, is written by Chris Miller, whose name can be found on *Transactor*'s list of contributing writers. Buddy comes on a "flippy," with a C64 version on one side, and a C128 version on the other. Actually, I should say "versions." There are C64 and C128 versions using BASIC editor source code, C64 and C128 versions using a separate editor, a C128 version which runs under the C-Power "shell", and a C128 Z-80 cross-assembler, which uses BASIC editor source code. The C128 BASIC editor versions include a simple search and replace function that allows you to specify whether you want any occurrence of the string or only a word, but unfortunately doesn't allow you to specify a line range. The separate editor allows a reasonable number of word-processor style directives, such as block-move, search and replace, as well as horizontal scrolling for lines longer than 40/80 characters.

Buddy has many more pseudo-ops than LADS. You can write code to be assembled at one address, but with labels at another address, which is useful if you are going to be writing disk buffer code, or code which is going to be moved from one address to another. You can store tables as bytes, words, PETSCII strings, or in Commodore screen code. Unofficial 6502 opcodes (but not those of the Z-80) are supported, and you can load or save parts of symbol tables between assemblies. If you have a 1571, you can link files faster, by using burst mode.

Conditional assembly is supported, so that the same assembly file can produce object code for the C64, C128, VIC-20, and maybe even the Atari 800. Perhaps the two most useful pseudo-ops are the .bas directive, which permits you to link BASIC and machine language, even using symbolic SYS addresses in your BASIC program; and the .out directive which permits you to output bytes using your own machine code. (.out could be used, for example, to write a boot sector, to program an EPROM, or to save code to a file in some format other than the standard one used by Commodore - e.g. with some sort of checksum, every so many bytes, as in "&" files; or without the two-byte address header, as in standard GEOS program files; or with some sort of encryption, for copy protection purposes.) If you are tired of coming up with label names for minor branches, you can use "+" and "-" to refer to forward and backwards references.

One minor complaint about the .bas pseudo-op: when you give a symbolic SYS address, Buddy automatically adds an extra space after the SYS token. This is unnecessary, and it should be left to the programmer to decide.

A more important deficiency of Buddy is the lack of a usable macro language. Buddy does include three pre-existing macros, which can: compare two indirect addresses, fill an area of memory, and move memory. In theory, there is room for five new user-definable pseudo-ops. Unfortunately, the manual shows how to define a pseudo-op that will print out a message on each pass. To make good use of these user-definable pseudo-ops, what you really need to know is how to evaluate an operand, and how to put a byte into the object stream, and this information is not in the manual. You do get a list of labels used in the assembler, so I guess it is not impossible to trace things down, but it won't be easy.

It can be useful to be able to program the Z-80 outside of CP/M, especially if you want to move memory around, and having a cross-assembler makes this possible. It might be nice to have this assembler support 8080 mnemonics, for those of us who have the disability of having used the CP/M 2.0 assembler for a while, and who, therefore, use Intel mnemonics except for the unique Z-80 instructions.

*It can be useful to be able to program the Z-80 outside of CP/M, especially if you want to move memory around, and having a cross-assembler makes this possible...*

The disk also includes C64 and C128 programs to convert BASIC editor source code to separate editor source code, and the source code for a disassembler.

**The magic of Merlin**

Merlin 128 is the latest incarnation of Glen Bredon's public domain (freeware?) Big MAC assembler for the Apple ][ series. A less expensive Commodore 64 version is sold separately. When you boot up Merlin 128 (in 80-column mode only!), you find a menu which permits you to load, save, get a catalog, and do several other things. You create code by going to the editor.

The editor has two modes: an "immediate command" mode, in which you give such commands as "ASM"; and a full-screen editor mode, which is used for all input of text. (The C64 version uses a line-oriented editor.) There are block move, find and replace, and other similar commands, along with a half-screen mode in which the ten bottom lines are frozen. Certain editor commands can be undone (a feature word processors should have!). Immediate mode also includes commands to move and copy lines, and to find or change words or strings in a line range, with or without confirmation of each change, and with a wild card character available in the find string. The editor automatically tabs lines into:

```
LABEL     OPCODE     OPERAND     COMMENT
```

format. If you don't have a label, you must leave one space.

The "ALT" key is used for keyboard macros. For example, ALT-a defaults to CONTROL-B/SPACE/LDA. Since Control-B goes to the beginning of the line, ALT-a puts "LDA" in the opcode column. The altkey definitions and several other features can be changed and saved to disk. Keyboard macros, by the way, are very fashionable just now, and there are all sorts of books and packages on disk for such programs as Lotus 1-2-3. The PBS program, "Computer Chronicles," just ran a show on keyboard macros, and one person had even written an adventure game in terms of Lotus 1-2-3 macros. A chacun son gout!

Merlin is a macro assembler. Macros are, essentially, templates for strings of assembly language instructions and/or assembler directives, into which you insert variables. For example, to do a whole bunch of two-byte additions, you could write:

```
DADD MAC
CLC
LDA ]1
ADC ]2
STA ]1
LDA ]1+1
ADC ]2+1
STA ]1+1
<<<
```

Then, if you wanted to two-byte-add TEMP to $1234, you could simply write:

```
>>> DADD.$1234;TEMP
```

Or, for example, let us suppose that you were writing a FIG-style Forth interpreter. The macro:

```
HEADER   MAC ]OLDPC = *
TXT      ]1 ]LENGTH = ]OLDPC-*
DS       -]LENGTH    ;BACK UP * TO ]OLDPC
DFB      ]LENGTH     ;BYTE WITH LENGTH ]1
DCI      ]1          ;LAST BYTE HAS HIGH
                      BIT TURNED-ON
DA   ]VLIST ]VLIST   = ]OLDPC
<<<
```

will maintain headers. For example,

```
>>> HEADER.'forth'
```

gives you the Power equivalent of:

```
.byt 5;length of "forth"
.asc "fortH";note H has high bit
 turned on
.wor last'header
```

except that the Merlin macro automatically computes the length of five, and keeps the address of last'header.

Here, the labels ]VLIST, ]OLDPC, and ]LENGTH are variables - labels which can be redefined. In Merlin 128, there are also local labels whose scope is only between two global labels.

Merlin also has conditional and loop structures, and Merlin 128 has a five-byte floating point pseudo-op, but it only takes integer arguments. You can get a printout of how many processor cycles are used by various routines, but there are inherent ambiguities - some ops take more time if a page boundary is crossed at run time, and there is no way to test this at assemble time. Merlin doesn't have the equivalent of Buddy's .out, .bas, or .scr. It does have one user-defined pseudo-op.

One deficiency of Merlin is that although it has many ways of defining text, none of them really corresponds to Commodore PETSCII. The TXT directive uses chr$(65) for "a", but chr$(97) instead of chr$(193) for "A". (There is a directive for turning on the high bit of text - if you use single quotes, you don't get the high bit; if you use double quotes, you do get the high bit.) The difference between chr$(97) and chr$(193) does not really matter if you are printing to screen, but it does make a difference when you scan the keyboard. You can use the user-defined pseudo-op for PETSCII strings, and, indeed, the disk contains the source and object code to do so.

*One version of Sourceror disassembles to memory and can handle about 6K of object code; the other uses the disk for temporary files, and can handle $7f00 bytes of object code...*

Merlin can link source files in several ways. You can maintain libraries of macros. Ordinarily, it assembles to a buffer at a fixed address, and you then save to disk from the menu. But if you don't have room to do this, you can assemble directly to disk. Both Merlin 64 and Merlin 128 have their own Apple-style monitors, and Merlin 128 also allows access to the more useful ROM monitor. Merlin 128 can also create relative object files, which can then be linked together at any given address. (You can specify in the source code of a relative file that an error condition will result if the object is loaded above a certain address.) Unfortunately, there is no standard format for relative object files on the C128.

If you have Joe Smith's Fortran compiler for the Atari ST, its relative object files will, most likely, be compatible with the relative object files put out by Digital Research's CP/M-68K relative macro assembler, and the same is true of CP/M-80, MS-DOS, AmigaDOS, Unix, et. al. I am, however, virtually certain that no two of the relative object files put out by Merlin 128, Abacus "Super C," Pro-Line "C Power," and Eastern Software's MAE 64 are compatible with one another. I doubt that anyone except Commodore itself could promote such standards, and don't hold your breath!

**Disassembly with Sourceror**

Merlin includes a labelling disassembler, Sourceror. This disassembler is more difficult to use than the disassembler which comes with Buddy, but it is likely to require less post-editing of source code. When you use Buddy's disassembler, you specify the area and bank of memory to be disassembled, wait a few seconds, and the source code shows up in memory, in BASIC editor format (with line numbers corresponding to the address in decimal). The disassembler will not stop when it hits a table of addresses, or text, and, therefore, you are going to have to edit this source code.

With Merlin's Sourceror, on the other hand, you load in a disk file (or choose an area of memory in bank 0 - you can specify an offset, so that the byte in $a000 really belongs at $2000) and then you go through the program, telling Sourceror where to disassemble, and where to interpret the memory as addresses, bytes, or text. If you make what appears to be a mistake, you can back up. Alternatively, if you hit some fill bytes that are irrelevant to the rest of the program, you can skip over them. There is a directive to interpret two bytes as an address-1, (which is very typical of 6502 programming - look at the BASIC ROM), or as an address in reverse order, high byte first. The disassembler will properly interpret 2C A9 01 as:

```
HEX 2C
LDA #$01
```

instead of BIT $01a9, but if it's an EOR instruction which follows the 2C, you'll have to do that by hand. Sourceror will also correctly handle both the Kernal and BASIC ROM versions of PRIMM. (This is probably because DOS calls under Apple ProDOS have a similar format of JSR to address followed by data.) There is a file of standard labels, such as CHROUT, which Sourceror uses to do its labelling. This list can be edited.

One version of Sourceror disassembles to memory and can handle about 6K of object code; the other uses the disk for temporary files, creates multiple source files, and can handle $7f00 bytes of object code. One suggestion I would make is that future versions of Sourceror have the option of making use of expansion RAM modules, which might speed things up. (Actually, what should be done is that the Kernal ROM should be rewritten to allow access to and organization of expansion RAM as a RAM disk.)

The disk also includes programs to generate cross references - showing where each source label is defined and referenced; to format a file - so you can use the editor as an inferior word processor; xand to print the assembly to a disk file, instead of to the printer. There is source code given for sample programs and routines, including a 1571 disk copier (which uses u1 and u2, instead of burst mode - so it could be faster) and a disk sector editor. The program comes on a 1571 disk, but no file goes past side 0. However, Merlin is the slowest and largest of the three assemblers, and to load it and then do a long assembly, with lots of linked files, would be *very* slow on the 1541.

**Head to head**

None of these assemblers use any sort of copy protection - perhaps because students of assembly language might regard copy protection as nothing more than a tutorial, to be disassembled, understood, and, of course, cracked. Or, perhaps, assembly programmers will empathize with the plight of the poor programmer denied his just royalties (more so than compiler programmers?).

All in all, I would say that Merlin is the most flexible of the three assemblers, with Buddy having certain advantages with regards to its .out and .bas directives, and LADS having the advantage of the complete source code being provided. As far as manuals is concerned, the LADS "manual" is the best, although it is not really a manual so much as a tutorial on assembly language. Merlin has a fairly complete manual, but it has no tutorial information on 6502 assembly language. The Buddy/Power manual, at least the one provided by Spinnaker, has the least complete information; as I have said, the documentation of user-defined pseudo-ops is useless. It does have a small, but well-organized summary of both 6502 and Z-80 assembly language.

**A few final thoughts**

1) Since the CP/M Macro Assembler can do 6502 machine language to some extent using a macro-lib file, maybe one could write 8080 or Z-80 assembly language in Merlin, using a macro file.

2) Back in January 1986, Yves Han wrote a C64 assembler for COMPUTE!'s Gazette, as an extension of the BASIC interpreter. Han poked the BASIC ROM into RAM, and then modified it, and added extra code. This is harder to do, of course, in the 128. Assembler op-codes and directives were tokenized, and you had to use a BASIC FOR-loop to get the three passes the assembler used. The advantage of this assembler was that you could use the full power of BASIC. For example, if you wanted to assemble the length of a$, followed by a$ reversed, you could just write:

```
100 byte len(a$)
110 for j= len(a$) to 1 step -1: byte
    mid$(a$,j,1) :next
```

Furthermore, having this as a subroutine, you could use this as a macro. This assembler is very fast, although it can't print out listings. I wonder if anyone else has considered an assembler where you could use BASIC operations within assembly language. (I notice that BBC BASIC, which runs on 6502-based Acorn computers, has an assembler built in, but I don't know if it allows this sort of manipulation.)

3) 6502 Forths generally include a version of William Ragsdale's public domain reverse Polish assembler. (e.g. 5 # lda, instead of lda #5) This is not a stand-alone assembler. Forth is a language in which programming involves defining new keywords in the language, and the assembler is used when you need to write a word which will perform more quickly than its pure Forth equivalent.

While this is only a one-pass assembler, designed for relatively small routines, it does have words to allow you to do a "while" loop, or an "if-else-endif" type branch. (The assembler saves the address of the branch instruction, and when it comes to the keyword which tells it how far to branch, it pokes that information into the branch instruction.) Since the assembler is written in Forth, you can always add new keywords to it in Forth; thus it is a macro assembler. (Forth uses a stack to pass parameters.) Has anyone thought of writing a two-pass, Ragsdale-like stand-alone assembler?

**window128:** *See article on page 49.*

```
KK 100 print "c128 window demo        jim butterfield"
CH 110 dim f(4)
FN 120 data 2,3,5,7
FC 130 for j=1 to 4:read f(j):next j
OL 140 print
OC 150 print "(pick factor range, eg. from 77 to 392)"
CN 160 print
IF 170 input "from";f
IA 180 input "to";t
DD 190 if t>f and f>0 and t<99999 goto 210
JJ 200 print " .. try again ..":goto 140
HH 210 f=int(f)
DE 220 print "{clr} from";f;"to";t
DK 230 for j=1 to 4
ID 240 print tab(j*6-5);" /";f(j);
MN 250 next j
MP 260 print tab(5*6-5);" /high";tab(6*6-5);" prime"
KC 270 w$=chr$(27)+"t{6 right}{10 down}"+chr$(27)+"b"
OF 300 for j=f to t
DE 310 if j<4 or j=5 or j=7 or j=11 goto 590
AA 320 for k=1 to 4
MM 330 s=f(k):d=j/s
IL 340 if d=int(d) goto 600
DE 350 next k
IO 360 k=5
HP 370 l1=sqr(j+.5):l=11
ND 380 if l1<11 goto 590
MP 390 for l=11 to l1 step 2
LJ 400 d=j/l
OP 410 if d=int(d) goto 600
MI 420 next l
CN 590 k=6
HN 600 print "{home}{home}{down}"
KM 610 print tab(k*6-5);w$
HN 620 print str$(j);
IF 630 next j
NP 640 print "{home}{home}{13 down}"
```

# A Tale of Two Cartridges

## *Final Cartridge III vs. Action Replay Mk. IV*

**Review by Noel Nyman**

The Final Cartridge III, $54.99

Action Replay Mk. IV, $59.99

(prices in U.S. dollars)

Both cartridges distributed by Datel Computers, 3430 E. Tropicana Ave., Unit #67, Las Vegas, NV 89121

These cartridges are aimed at different audiences. The FC-III is billed as "a powerful 64K ROM-based operating system for the C64 and C128." The emphasis is on "keep it simple" for the beginner, while giving all users extra features such as a notepad and calculator available as 'windows.'

The Mk. IV is advertised in full page, four colour magazine ads as "the ultimate backup cartridge." The drawings show a futuristic space vehicle cartridge "docking" at the C64 expansion port.

In spite of the differences in marketing, the cartridges have much in common (see the comparison chart accompanying this review).

### Final Cartridge III

The FC-III comes in a box nearly the size of the C64 itself, slickly printed with color pictures of various computer screens and descriptions of features. A carrying handle is provided, in case you find the need to tote a two by three inch cartridge around in a eight by twelve inch box.

The large box is necessary for the documentation, which should win an award for originality of format. The pages measure about eight by twelve. They are loose in the box, and punched with two holes on the eight inch side. Metal tabs are provided to hold the pages together.

Each piece of paper has four document pages printed on it. If folded in half, the collated pages would make an average sized booklet. Unfortunately, if you did that with the FC-III docs, the page numbers would be scattered all over the place.

The documentation is clear and easy to read, though occasionally brief. Some commands described are not mentioned in the summaries, which makes looking them up difficult. There are a few awkward phrases, typical of manuals translated from one language to another (FC-III originates in the Netherlands). Some of the diagrams do not correspond to the actual screen displays. More on this later.

The cartridge is the usual black shell with a few surprises added. There are two square switch buttons identified as "reset" and "freeze." A small red LED lights when the FC-III is active and goes off when the cartridge is 'killed.'

On power-up, you see a menu bar across the top of the screen in place of the usual start-up screen. This bar controls several pull-down menus which, in turn, control the cartridge options.

You select one of the bar options by 'pointing' at it and 'clicking.' The pointer on the FC-III is a coloured arrow. You move the arrow using a mouse, a joystick, or the function keys. Once you've pointed at an item, you select it by 'clicking' ...pressing the mouse button, the joystick fire button, or the Commodore logo key on the keyboard.

The choice of the function keys instead of the cursor keys to move the pointer is frustrating. I kept hitting the cursor keys out of habit. I often overshot my target when using a joystick. The pointer speed and acceleration can be changed. But it can't be slowed enough to let joystick novices such as me operate efficiently.

### Windows arrive

FC-III brings 'windows' to the C64, common in other computer operating systems such as the Amiga and Macintosh. A window is a box that appears on the screen, and is used to perform an operating system function. If other information is already on the screen, the current or active window overlaps it. You can move windows around and display several at one time, although only one will be active. When you've finished with the function controlled by a window, the window disappears and whatever screen information was under it is restored.

**Cartridge comparison chart**

| | MK-IV | FC-III |
|---|---|---|
| Window driven | | * |
| Clock/alarm | | * |
| Calculator | | * |
| Notepad | | * |
| **Printing** | | |
| Low -res screen | | * |
| High -res screen: | * | * |
| "negative" print | * | * |
| change size | | * |
| print sideways | | * |
| print sprites | | *(1) |
| **Disk operations** | | |
| Fast load | *(2) | * |
| Fast save | * | * |
| Fast format | * | * |
| "Wedge" commands | * | *(3) |
| File copy | * | |
| Disk copy | * | |
| **Freeze** | | |
| Backup memory | * | * |
| View sprites in memory | * | |
| Save sprites | * | *(4) |
| Change sprites | *(5) | |
| Disable collision det | * | * |
| Swap joystick ports | | * |
| Add "autofire" | | * |
| Save screen to disk | * | |
| Poke memory | *(6) | *(6) |
| Modify text | * | |

**Notes**

(1) In some games, sprites definitions are changed part way down the screen using a "raster interrupt" routine. Depending on the raster location at the moment of freezing, some sprites may not appear on the hard copy.

(2) The MK-IV offers two fast loading routines; see text.

(3) Some of the standard wedge commands are available in the FC-III by using unusual syntax. Sending DOS commands from BASIC follows the Simons' BASIC syntax. For example, to see a disk directory, you type: DOS "$

(4) The FC-III can save any portion of memory with the MLmonitor. But, the user must identify the starting and ending addresses for the sprite.

(5) Limited sprite editing: flip, erase, mirror, and reverse.

(6) "Pokes" or parameter changes can also be made through the ML mon-

The advantage of windows is that a computer novice can operate the system quickly. No need to learn complex commands like `load "$",8` to view a disk directory, for example.

The disadvantage of windows for the experienced user is that you have to use the pointer to access everything, even if you know faster methods. Another disadvantage unique to FC-III windows is that they are only available in the 'desktop' mode. You can't call the window functions from a BASIC program or in direct mode from the BASIC screen. When you exit to 'desktop' from BASIC, you lose anything in BASIC memory.

Instead of icons (little drawings that somewhat resemble the options they represent), the FC-III uses mostly 'gadgets,' small blocks with real words inside them.

To read a disk directory, you point to the "utilities" option on the menu bar and hold the 'clicker' button or logo key. Five options are displayed, including "disk." Move the pointer down until the disk option is highlighted. Releasing the clicker causes the Disk Operations window to appear on the screen.

The window, which takes up about half the screen, is filled with gadgets. The largest one is the window's top border, a series of horizontal lines. By pointing at this gadget, holding the clicker, and moving the pointer, you can 'drag' the window to any location on the screen. This is typical of windows as used on other computers.

There are three DIR gadgets, each with an '8' and '9' gadget. To get a directory from drive 8, point to and click on the '8' gadget for one of the DIR's. The '8' will be highlighted. Then point to the DIR and click. A new window will appear, with the disk name and the first ten directory entries listed. Two arrow icons let you scroll up and down through the directory.

At the bottom of the new window are two additional gadgets, SORT and READ. The SORT gadget is not mentioned in the documentation, and the diagram of the directory window does not show it. If you click on SORT, the READ gadget is replaced by LINE. The files' listing order doesn't change.

If you select a file name by pointing/clicking on it, then click on LINE, you'll be given the option of adding a line above the selected file name. Now, when you click on SORT again you'll be asked if you want to write the directory back to the disk. When I tried that, all the windows vanished, the computer locked up, and I had to reset with the cartridge switch. Perhaps there is a way to sort the directory with the undocumented SORT gadget, but I couldn't find one.

The READ gadget is used to re-read the directory when you change disks. An alternative is to change disks, then call up a second window using a different directory window. The second window will overlap the previous windows.

The "exchange" icon (a bright square overlapping a dark square) is used to exchange the top, 'active' window with the last active window. The window now on top becomes the current window (usually, see the exception below). Or, if the option you want on an inactive window is visible, clicking on it will sometimes make its window active and initiate the action.

The "de-select" icon (a bright square with a dot in it), is used to remove a window from the screen. This does not remove it from memory, however. If you re-activate the window the old information re-appears in it.

This may be an advantage... keeping old directory listings handy in memory for future reference. But for most of us, it adds another point/click sequence on the READ gadget unnecessarily. The only way to completely initialize the windows is to exit to BASIC, then return via the DESKTOP command, or reset the cartridge.

### More disk operations

From the Disk Operations window, you can execute many common disk commands, as well as a few new ones. "Fast format" takes about 28 seconds. It gives you a prompt window, reminding you that this command will wipe out existing data, with the option to abort. Again, the pointer must be moved and clicked to either continue or abort the format.

When you select fast format, the name of the last disk read appears in the "from" window. If you don't select that window and type in a new disk name and ID, the disk will be formatted using the name and ID of the last disk you read.

**LOAD COMPARISON TIMES**

| | no cart | FC-III | Mk. IV Turbo | Warp |
|---|---|---|---|---|
| Flight Simulator, to Miegs field (size in blocks) | 3:35 | 0:30 (196) | 0:24 (199) | 0:12 (211) |
| Disk Maintenance, to main menu (size in blocks) | 1:08 | 0:19 (138) | 0:16 (133) | 0:08 (141) |
| Easy Script, to document in place (size in blocks) | 0:57 | 0:16 (88) | 0:11 (67) | 0:07 (71) |
| BASIC program that loads data (size in blocks) | 1:49 (145) | 0:17 (107) | 0:12 (101) | 0:07 (106) |

"Empty" performs a 'short New,' which reformats the directory to 'erase' all disk files. You can also rename and scratch files, validate or rename the disk, and read the disk status. For most of these commands, you'll need to select the appropriate gadget, then select the DO gadget to perform the command.

RUNning a file can be more complex. First, you select the file to be RUN by pointing/clicking. Then you click on the RUN gadget and the DO gadget. FC-III will pop up a new window reminding you that you'll leave the desktop. If you click on the RUN gadget in the new window, FC-III will switch to BASIC mode and LOAD and RUN the file.

As a challenge, FC-III does not identify the file types in the directory. This makes it easy to try to RUN sequential files. The system doesn't check for this possibility (assuming that you've memorized all your disks' file types, I guess), and will try to RUN the sequential file. The user is left with a flashing light on the disk drive and a "?FILE NOT FOUND ERROR" message on the BASIC screen. Very cryptic indeed for the novice, who will be frustrated by the fact that the file obviously *is* there in the directory window!

Also, if you have several directory windows, you can highlight a file in any of them for the RUN command. But, you'll get a "no file selected" message if the DIR window you've used isn't the active one in the Disk Operations window.

### Other FC-III window options

Another window lets you set a system clock in either 12 or 24 hour format and add an alarm as well. The docs don't tell you that you can't set the alarm unless you've already selected alarm mode in the clock pull-down menu.

A simple calculator window is also available. It does four-function math with a memory. Fortunately, you can enter numbers from the keyboard rather than by pointing/clicking their gadgets, although that works too. The calculator does not include square root, per cent, or memory +/- as shown in the picture on the front of the FC-III box.

Another window lets you change some of the desktop preferences, such as screen colours, pointer device port, and pointer speed. The BASIC preferences window lets you add a keyboard click, key repeat, defeat cursor blink, change the default device, and use the numeric keypad on a C128 in C64 mode. There appears to be no way to save the new preferences for future sessions. They have to be changed to your liking each time the computer is turned on. With all the pointing and clicking required, I soon settled for the default values.

The Notepad option removes windows and gives the user a blank screen with a new set of pull down menus. Text can be entered and edited using the regular cursor keys. You can control the line and character spacing, and select bold and proportional print, and wordwrap. The text is not limited to a single screen, and the resulting file can be SAVEd to disk or tape and recalled later in Notepad mode. It can also be printed.

The FC-III supports Commodore, RS-232 serial, and Centronics parallel printers. The latter two types require a special cable "available from your dealer." That may be true in Europe, but here, the few local Commodore dealers knew nothing about any special cable. But it is available mail order for $20 US.

I tested the cartridge with a Star Micronics SG-10 and a Micro-Graphix MW-350 interface set for Commodore emulation. This combination usually works well as a 'Commodore' printer. The result was a plain vanilla printout of the text... no proportional spacing. The text did word-wrap properly, in 80 columns rather than the 64 columns used for the screen display. The printer locked up when I tried printing in bold.

The Notepad text files are stored as PRG types in PET ASCII. The various options such as proportional print and line spacing, are screen options only and are not SAVEd with the text. Text files from other sources can be brought into the Notepad, provided the file is type PRG. Since the first two bytes of a program file are the load address, Notepad automatically removes them. If your file was originally a sequential type, the first two text characters will be discarded instead. There's no way to view a disk directory from Notepad mode.

## FC-III BASIC additions

The FC-III adds several features to BASIC. LIST scrolls up as well as down. A Shift-L in a REM normally defeats LISTing. Shift-L with the FC-III in place LISTs as OFF, so the balance of the program will LIST properly. The OFF command is documented as part of the TRACE OFF and BAR OFF options. But, used by itself, it appears to be equivalent to KILL, which disables the cartridge.

Also handy are the DUMP command, which displays the values of all simple variables, and the ARRAY command which does the same for arrays, including string arrays. The DUMP command has trouble with DEF FN's. It confuses them with integers and uses the bytes that point to the definition address (low/high order) as an integer value (high/low order).

The MEM command shows the total BASIC bytes available, and how many are used by program, variables, arrays, strings, and how many are free. Although the documentation claims that FC-III adds "24K bytes extra memory in BASIC," the MEM command reveals only the standard 38,911 bytes available.

TRACE displays each line of BASIC at the top of the screen as it's executed. TRACE can be slowed to a viewable speed using the CTRL key, although this will interfere with INPUT commands. TRACE uses address 682 as a flag. Any non-zero value here will cause TRACE to be active. Some machine language programs are stored in this area, and their presence may trigger TRACE mode at inappropriate times.

AUTO also uses low memory. Location 681 contains 64 when auto is active. Locations 820/821 hold the last line number used and locations 822/823 hold the increment.

Programmers using FC-III should avoid locations 679-767 and 820-827 since other cartridge functions may use these memory areas.

PACK and UNPACK are the most curious BASIC add-ons. PACK changes BASIC so it lists as:

```
1987 SYS 2061
```

The command moves the entire BASIC program up in memory, and adds a machine code routine to the beginning of the program. When RUN, the SYS command copies the BASIC program back to the normal starting location, changes a few vectors, and RUNs the program. Machine language programmers may want to examine the code used to do this, since it must overwrite itself to move BASIC back down.

The program is simply put back in BASIC space to RUN. If RUN/STOP is used, the entire program can be LISTed normally. This makes UNPACK unnecessary.

MON enters a machine language monitor with all the usual commands. Added commands allow editing memory as sprites or text characters, and viewing common memory areas as RAM or ROM. The monitor can operate on memory in the disk drive as well as the computer.

BAR allows displaying a menu bar at the top of the screen. This can only be done with a joystick or mouse button. The menu bar shows the definitions FC-III gives the function keys. There appears to be no way to change these. Other options include switching to the monitor, the desktop, or the freezer menu.

Two pull-down menu options list the added BASIC commands, including some not documented. HELP doesn't generate any errors, but doesn't seem to do anything either. REPLACE is more interesting. It uses the syntax

```
REPLACE  old,new
```

where "old" and "new" can be any variable name, BASIC command, or text in quotes. Any lines changed are listed as REPLACE executes.

BAR OFF disables the BAR for programs that object to its presence.

### Disk commands

The 'dos' command is used as a "wedge", acting like the ">" or "@" character normally used. Giving the command 'dos' by itself will read the error channel.

```
dos ''command''
```

sends the command to the disk drive. For example, DOS "$" reads and displays the disk directory to the screen.

DLOAD and DSAVE perform disk LOAD and SAVE with no need for a ",8" after them. DLOAD is *not* the same as its C128 (or BASIC 4.0) namesake. It performs a BLOAD instead, placing the program at its original location address. Two similar commands are DVERIFY and DAPPEND. All of the 'D' commands can be entered as 'D' followed by the next letter shifted.

APPEND (or DAPPEND) is used to add a program from disk to the program in memory. The usual stipulation is that the second program must have higher line numbers than the last line of the first program. To 'avoid' this situation, FC-III includes the ORDER command, which moves lower line numbers from the appended program into numerical sequence with the original program lines.

This creates a sort of merge. If the new lines make sense when executed with the old, the program will RUN. If both programs have the same line number, the new line is placed in sequence ahead of the old one. But, the old line is still retained in memory, and continues to execute. Editing, GOTO and GOSUB will see only the first, or newly added, line. This makes for strange and sloppy BASIC.

### The Action Replay Mk. IV

In contrast to the slick packaging of the FC-III, the Mk. IV came via mail order in a plastic bag inside a plain brown cardboard box. The cartridge shell is red plastic, with no labels of any sort. Two round black switches stick out the back at the right.

The documentation is a saddle-stitched booklet, using good quality typewriter printing. The instructions cover three versions of the cartridge.

At power-up, you see a four-option menu. Selection is done with the function keys, and there are no windows. F1 fills memory with a known value. This aids in making frozen copies. F3 creates a 'normal reset', disabling the Mk. IV for software that won't load with the cartridge active.

F5 calls up a utilities menu that executes DOS commands. The format routine takes only 13 seconds. The menu also accesses a full disk copier and a file copier. The latter looks remarkably like Jim Butterfield's "Copy-All." This menu also saves a special fast loader program to disk (more later).

The only exit from this menu is to "Fastload," the same as the final option of the first menu.

### Mk. IV BASIC mode

Except for an added message and a change to white text, the Fastload screen looks like the standard C64 screen. No windows, no notepad, no calculator.

You do have the usual disk wedge commands with the usual '@' syntax. @$ displays the directory, for example. The wedge uses the 'last device accessed' memory location 186, which can confuse it after printing; the @$ command will try valiantly to get a directory from the printer. A @8 sets things right again.

The '/' entered in front of a file name will load the program. This is often done after listing a directory. As with the Epyx FastLoad cartridge, if the file is larger than nine blocks, you must space over the remaining digits.

```
/5 "FILE"   PRG
```

will try to load a file named "5."

BOOT will load a program, then SYS to the starting address of that program. MERGE is a true merge, rather than APPEND, which is also supported.

The line numbers of the program being MERGEd from disk can be changed as part of the MERGE command.

The AUTO command has a small bug. If you edit a logical line that ends with a number, and the number wraps to the start of the next physical line, AUTO turns off.

My favorite is LINESAVE, which has the syntax:

```
LINESAVE "filename",8,startline-endline
```

and SAVEs a part of the BASIC program in memory.

COPY and BACKUP activate the built-in file and disk copying routines.

The Mk. IV has some of the BASIC commands found in the FC-III (see the comparison chart), although the FC-III is the clear winner in BASIC additions.

The machine language monitor is virtually identical with the monitor in the FC-III, but does not directly edit sprites or text characters. It does display memory as CBM screen codes, as well as PET ASCII text, hex values, or assembly code. It will also display and allow changes to the I/O registers.

### To freeze or not to freeze

'Freezing' is a technique for halting a program in progress. A copy of the frozen computer memory can be saved. If that memory 'snapshot' can be successfully reloaded, the program will continue from the point at which it was halted. See "How They Work" at the end of this article for the technical explanation of how this is done.

I first saw this as strictly a 'pirate' technique. Since the program is fully loaded when halted, all the copy protection on the disk is bypassed. The memory snapshot can be copied with any file copier and passed around freely.

But there are legitimate uses for freezing. Most frozen copies take less time to load than the originals. Many commercial programs are incompatible with fast loading routines, and with them the improvement can be dramatic. Since the copy protection isn't used by the frozen copy, there's none of the head banging found on older commercial programs.

If you use several options of the same program, you can freeze a version of each, after loading the necessary overlays, or selecting menu options. If you're a game player, you can freeze the game just before taking on the next adversary. If you loose, you needn't fight your way back through the lower levels again.

*Transactor* does not condone theft of software by any method. Use of freezing cartridges on any programs you don't own is unethical. But legitimate owners of programs should know about all the options they have for making the best use of their software. We're not telling the pirates anything new here. They already know all about freezing programs.

### Freezing with the FC-III

The FC-III freezer menu can be accessed from pull down menus or by pressing the "freeze" button on the cartridge. The latter may be the only option if a program won't load with the cartridge active. In that case, you use the KILL option to disable the cartridge, turning the red LED off.

The freezer bar controls pull down menus to back-up the program in memory, change some game options, change screen/border colors, and print screen dumps. To freeze, just select "disk" or "tape" from the backup menu. Only the joystick or mouse will point/click on the freeze menus.

The frozen memory is saved to disk as two separate files, "FC" and "-FC." You can rename these files later. There's no message given about file size before the save starts. You should use a reasonably empty disk since frozen memory often takes 150 disk blocks or more.

You're returned to the desktop menu bar with no option to resume the frozen program.

**Freezing with the Mk. IV**

The freezer menu is accessed by the freeze switch. The switches aren't labeled, and if you push "reset" instead, you'll have to start over. From this menu, you can enter the monitor, get a disk directory, do some interesting sprite manipulations, save or print the hi-res screen, or back up the program.

Before the backup is saved, you're given several DOS options, including "directory" and "format." You're also told the size of the file to be saved and given three save formats: standard, turbo, and Warp*25. Files saved using "turbo" are standard DOS files with an optimized "interleave," the number of sectors skipped between two consecutive program sectors on the disk. Files saved in this way will load a bit faster with the Mk. IV or its fastload routine. They will load without the cartridge also, but more slowly.

The Warp*25 uses non-DOS techniques to save files. They can be re-loaded very fast with the Mk. IV in place, or with a special loader program that's saved into the directory track on the disk using a special menu option. The Warp*25 files are larger than turbo files. They require consecutive tracks on a disk, so may report a "disk full" error when many free blocks remain. The Warp*25 files load much faster than any others. In fact, Warp*25 may be the Mk. IV's best feature. See the loading times comparison chart.

**Hi-res screen options**

The Mk. IV allows you to freeze the program, then save the current hi-res screen in one of six software formats: Blazing Paddles, Koala, Advanced Art Studio, Artist 64, Vidcom 64, and Image System. Only multi-colour hi-res screens can be saved.

Freezing may occur while the "raster" (the part of the video system that 'paints' the image on the screen) is mid-screen, resulting in strange images. The screen can be viewed with the F7 key before the save. You can restart the program and 'jog' through a picture to try for a better image. Stationary hi-res screens are easy to capture. Sprites will *not* appear on the saved image.

The FC-III does not have a screen save option. You can view the captured hi-res screen for printing purposes. Sprites are visible on the FC-III captured screen, although the presence of some moving sprites will depend on the raster location.

You can select among printer types, print density, the size of the finished product, and sideways printing. With my printer/interface, the various density options produced the same output. Size can be increased both vertically and horizontally. The default values are both '1,' which produces an output about three by five inches. Increasing the horizontal value to '2' makes the hard copy twice as wide - a bit wider than a nine inch printer can reproduce. You can make a full size '2' by '2' image by specifying the "sideways" option.

In the larger sizes, picture elements become larger and the gray scale hatching detail increases. The hard copy always starts with a corner of the screen. So it is difficult to get a large scale hard copy of objects in the center. You can enter the freeze mode to print low resolution screens with the same options.

You are returned to the desktop menu bar after printing, with no option to restart the program.

**Gaming options**

Both cartridges offer several features of interest to game players. Both will disable sprite collision detection. This may allow you to survive unscathed in some games, although both manuals caution that many games do not use the built-in C64 sprite detection systems. Both cartridges make the machine language monitor available in freeze mode, so changes can be made in the program. The Mk. IV makes this easier with a POKE mode.

The FC-III adds an 'autofire' feature to the joystick fire button. It also will 'swap' joystick ports in case you start with your joystick in the wrong place. Actually, this command changes the keyboard scan too, with fascinating results on programs such as Flight Simulator.

The Mk. IV will search for and replace PET ASCII text in memory. I was able to change "player" to "sucker" in Beach-Head without the need to hunt for the text with the ML monitor. It will not change hi-res text, of course.

A real plus for the Mk. IV is its sprite mode during freeze. Memory can be displayed as sprite images, either hires or multi-color. The sprites can be saved to disk individually, and loaded back into memory. They can be wiped out, inverted, mirror imaged, or flipped. The sprite address is displayed, facilitating the use of the ML monitor to save a range of sprites.

I extracted the rotating gun from Raid on Bungeling Bay for use in my own BASIC program. Then, I replaced the gunboat sprites with tiny Jump Men. This is great fun. But it would be more useful if the Mk. IV included a sprite editor.

**Compatibility**

Frozen FC-III programs use a fast loader routine which conflicts with the Mk. IV. The turbo or standard Mk. IV frozen programs can be loaded with the FC-III.

The Mk. IV claims to be compatible with the 1541, 1541C, 1571, and 1581 disk drives and the C64, C128, and C128D. The FC-III works with the C64 and C128. No list of disk drives is provided. I tested only the 1541.

The Mk. IV command prompts and menus are printed in white. The SX-64 background screen color is also white. So, using the Mk. IV with the portable becomes more of a challenge. You can change the text color from some modes. The FC-III prints in text color, so all messages are visible. But the hires windows and menus are very hard to read on the built-in monitor.

**Final comparison**

So, which cartridge should you buy? If your goal is a simpler operating system, the FC-III is the clear choice. The window environment provides a (dare I say it) "user friendly" atmosphere, free of arcane commands. Experienced users will quickly skip to the BASIC command screen instead.

The FC-III is also the best for screen dumps to the printer, both for options and its ability to display sprites.

The Mk. IV gives game players more options, and wins the fast loading race hands down. If windows don't interest you, the Mk. IV is the better value.

**CAPSULE COMPARISON (zero to three stars)**

| | | FC-III | Mk.IV |
|---|---|---|---|
| Features | | | |
| BASIC commands | : | *** | * |
| ML monitor | : | *** | *** |
| Screen dumps | : | *** | * |
| Screen saves | : | - | *** |
| Gaming | : | * | ** |
| Disk operations | : | ** | *** |
| Freezer operation | : | *** | *** |
| Windows | : | ** | - |
| Ease of Use (beginner) | : | *** | ** |
| Ease of Use (experienced user) | : | * | *** |
| Overall rating | : | ** | ** |

## How They Work

The memory in the C64 and C128 is shared by several systems. The MPU (Micro Processor Unit) stores information in memory for use as program data and for screen display. The VIC (Video Interface Chip) looks at memory also to get the information to create a screen display.

Only one device at a time can access memory. The VIC chip has priority, and uses a technique called DMA (Direct Memory Access) to assert its rights to RAM (Random Access or read/write Memory). When the VIC chip activates a special DMA circuit, the MPU is put on hold until DMA is released.

The DMA line is also available at the cartridge or expansion port. Freezer cartridges produce their own DMA signal when the freeze button is pressed. This halts the MPU.

A new processor inside the cartridge takes over. This second MPU has its own operating system in ROM inside the cartridge. It usually has some RAM in there also.

The cartridge MPU can read the keyboard, joysticks, disk drives, and make changes in the regular computer memory. On command, it makes an image or 'snapshot' of the computer RAM on disk or tape.

You can examine the innards of either cartridge easily with the aid of a small Phillips screwdriver. The usual snap tabs are missing from the shells, and the cartridges come apart easily. The FC-III is a complex system of integrated circuits. The Mk. IV is simple by comparison, owing largely to the custom SMT (Surface Mounted Technology) MPU/ROM chip mounted on the underside of the board.

Be careful in handling either exposed board. Both use MOS chips, which can be easily damaged by improper handling. Also, note that the switches in both units are simply soldered to the circuit board with no other mechanical support. Push gently on those switches. Heavy use may break the solder joints or the boards themselves. The exposed long metal sections of the Mk. IV switch may be more susceptible to damage than the black buttons of the FC-III.

# BrainStorm, BrainPower, Story Writer

*Word and outline processing for the C128*

**Review by Marte Brengle**

BrainStorm/BrainPower
Outline Processor/Word Processor for the C128
$22 US for both programs

Story Writer
Easy-to-use story-writing aid, $12 US

Available by mail-order from:
Country Road Software
70284 C.R. 143, Ligonier, IN, 46767 (219) 894-7278

Many people have trouble getting organized when they sit down to write. Students taking essay tests are often told to make an outline first, in order to help collect their thoughts and put them in some kind of logical order. That's not always easy to do, since many of us don't think in 'outline form.'

With the invention of word processors, the physical act of 'getting words on paper' became easier, but until recently there weren't many programs that could help with the organization of the words ahead of time. Now writers can use an 'information processor' or 'outline processor' like BrainStorm, and make the whole process amazingly easy and painless. I've used the outline processor in Microsoft Word, a far more expensive and complex program, and I believe that BrainStorm is equally powerful, and certainly easier to understand.

The BrainStorm disk contains BrainStorm 2.0 and BrainPower, a small but powerful word processor. You can choose which program you want from the colourful opening screen. Note that both programs are for the C128 only, in 80-column mode only.

If you choose BrainStorm, the first thing you'll see is a listing of the function keys for the program. These keys work from anywhere in the program, and let you manipulate files, check your outline for 'logic,' and change from one program to another. You can bring back a listing of the function keys from within BrainStorm by pressing the letter F.

The idea behind a 'brainstorm' is simply to put down as many ideas as you can, up to the program's limit of 100. They don't have to be in any order, or related to each other in any way. The program gives you a yellow "brainstorm box" to type in, which can contain up to 240 characters. Your words will scroll across the screen as you type. There are a few rudimentary editing controls available in the brainstorm box; the delete key works as you'd expect, but you toggle insert mode by pressing cursor-up. Pressing CTRL with the left or right cursor key takes you to the beginning or the end of a line. CLR clears the whole box, and TAB moves you across it ten spaces at a time.

Once you've written down everything you can think of, you move to the next phase of the program, in which you begin to put the ideas in order. The program will show you the first idea on your list, with the second one displayed beneath it. You can then compare the two and decide whether they are similar or not, skip to other ideas for comparison, delete ideas you don't want to keep, and move back and forth among the ideas at will. Eventually, you'll have compared all the ideas to each other and put them into groups according to similarities. Even then, though, your decisions aren't cast in stone. You move on to what's called the "trunk menu" in order to turn your grouped ideas into an outline.

The "trunk menu" presents your ideas under headings that initially are called "Group 1," "Group 2" and so forth. Each idea is preceded by a dash. At the top of the screen is a listing of all the available commands. You can get additional help by pressing the Help key, and the BrainStorm function keys are also operational here.

Now you begin to edit your outline, to put it into a logical order. Each individual idea is dealt with as a whole, and can be highlighted in colour by pressing cursor-up or cursor-down till you reach the one you want to work with. From there, you can move the idea to another spot in the outline, or edit it, or delete it, or add more ideas that go along with it. You can also move or delete entire groups of ideas by highlighting the "Group" heading.

As in the brainstorm box, each idea may be up to 240 characters long, although the whole idea might not show on the screen. If you'd like to see each idea in its entirety while you're editing, you can use the Extend command. This will display longer ideas as short 'paragraphs' so that you can work on them, but as soon as you're finished with the extended idea and move on, the screen will show only a portion of it again.

As you edit the outline, you'll want to indent some ideas to make sub-headings. One really nice feature of BrainStorm is that each level of indentation is in a different colour. This makes it very easy to check your outline to see if you've subdivided it properly. (There is a provision for monochrome screens as well.) As you move ideas around, you may have to work a little bit to get them indented or 'outdented' to the positions you want them to assume. Sometimes the program won't let you delete or change the indentation on a particular line unless the next line beneath it is at an equal level of indentation. You may have to change that just long enough to deal with the preceding line, and then change it back.

BrainStorm uses some familiar word processing commands. You can search for a particular text string, mark and return to your place, delete ideas singly or in blocks, and retrieve them from a 100-item buffer if you change your mind. In addition, the program lets you 'collapse' the outline, so that only certian levels are visible on the screen. If you have a very long outline, this can make editing much easier. Those levels aren't erased; they're just not visible on the screen at the present time. Once you've collapsed the outline, you can work on any particular level, even ones that aren't currently visible, by using the "show" command. This will open up the area you select, leaving the rest of the outline collapsed to save space. Once you're done, you use the "hide" command to collapse that particular area again. You can also use "hide" to selectively collapse a particular section of the outline. Any level that has hidden subdivisions will be preceded by a plus sign, so you can tell where they are.

Each section of the outline can also be edited separately by going to the "branch menu." This alows you to deal with one chosen area at a time without having the entire outline on the screen.

When you've finished with your outline, it's a good idea to press F2 to check its 'logic.' This will tell you if it's in proper outline form (all the headings and sub-headings in order, no "division by one" errors, and that kind of thing). You can also arrange your outline according to any particular logical strategy you prefer (the screen gives some help on doing this). I'd advise switching to the monochrome screen before entering 'logic,' though, because otherwise the text appears in black on a dark grey background and is very hard to read. All errors in logic will be pointed out when you return to your outline - there will be flashing lines in the appropriate places.

After you've corrected the errors, you can view your outline on the screen by pressing F3. This will give you an idea of how it will look when it's printed out. You can change the page breaks if you like, and check to see if your outline needs any further editing.

Once you've finally finished with your outline, you can print it out or save it to a disk (or both, of course). Since BrainStorm adds a "-." prefix, your filenames can't be more than 14 characters long. The files can be saved as ASCII files if you put an exclamation mark at the end of the filename. The program also gives you a choice of making automatic backups of your files. This means that if you happen to save a file with the same name as one you've already used, the old file will be preserved on the disk, with a .BAK extension on the filename. The finished outline can then be used with BrainPower, the word processor. (The program disk contains a good example of a BrainStorm outline: the outline for the entire program manual is there, and it's interesting to compare it to the finished manual.)

**BrainPower**

BrainPower is a small but powerful word processor, which you can enter directly from within BrainStorm. One of its main strengths, of course, is the ability to interface with BrainStorm. You can merge your outlines directly into your text, or you can have the outline in a window at the top of your screen and scroll it from point to point as you write. This makes sticking to your outline especially easy.

BrainPower is not a WYSIWYG word processor. Whether that makes a difference is, of course, a matter of individual preference. People who are accustomed to using SpeedScript, for example, will find the interface familiar. People who have been working with Pocket Writer may find that BrainPower takes a bit of getting used to.

The screen can be customized to some extent. It comes up with white letters on a black background, and the background colour can be changed. The author suggests trying a blue background if you're using a colour monitor. I found that a bit too eye-boggling, and settled for dark grey instead. It would be nice if the character colour could also be changed.

All the most commonly-used word processor commands are available. The cursor can be moved rapidly by pressing CTRL in conjunction with the cursor keys at the top of the keyboard. In addition, you can move ahead one word at a time by pressing the Commodore key and cursor-right. There is no equivalent 'move back one word' command, though. Insert mode toggles on and off. Alternatively, you can insert five or 20 blank lines at the press of a key. Block move and delete, search and replace, reformatting by paragraph or "format all," change case, justification, file linking, and headers and footers are all available and simple to use. You can also mark your place and return to it, which is very convenient when you're editing text. The commands are invoked by pressing the function keys or a CTRL-key combination, and pressing the Help key at any time brings up a series of help screens that explain just about everything the program does.

The program also includes some interesting features that are not commonly found in Commodore word processors, even those that are far more complex. You can bring up a simple four-function calculator, for example. You can view any file on a disk, including program files, although only the BrainPower files can actually be loaded. Pressing F5 to "load file" also lets you rename or scratch any of the BrainPower files on your disk. And, as an extra treat, you can hear J.S. Bach's Invention #13, a brief segment of which is played when BrainPower first starts up.

Formatting commands are imbedded in the text, and this is where people who are accustomed to WYSIWYG word processors may have to make a few mental adjustments. The program comes with preset printer codes for underlining, boldface, etc., but you may have to adjust those to match your particular printer. (Check your printer manual to find the appropriate values. The BrainPower manual gives clear instructions for assigning the codes.) The printer codes are imbedded in the text by pressing the Commodore key in conjunction with a letter key, and the instruction will appear in reverse video on your screen. If you don't find the pre-arranged printer mnemonics to your liking, you can change them to whatever you wish.

The program boots up with certain default values for top, bottom, and side margins. You can change them to suit your own preferences and save that configuration. Changing the format of the page is done by pressing the left-arrow key at the beginning of a new line and then typing in the appropriate values (again, people who are familiar with SpeedScript will find this procedure quite familiar). You can then reformat the document's appearance on the screen by pressing F3 or F4, if you wish. One quarrel that I have with the program is that no matter where you set your left margin, the text appears "flush left" on your screen. You could have a left margin of 50 and it would still look like it was at zero, but your lines would be very short. And the column number at the top of the screen refers to the cursor position relative to the left margin, not to its actual position on the screen.

BrainPower saves files in SEQ format, with a "w." prefix. When you press "load file," you get a window on the screen that brings up all files with the appropriate prefix. I have found that even if you rename a SEQ file from another word processor and give it the appropriate prefix, BrainPower will not load it. BrainPower files will, however, load into other word processors that accept SEQ files. The 'generic' geoWrite Text Grabber will work on them as welll.

As far as I can tell, neither BrainStorm nor BrainPower takes advantage of a RAM expansion unit, and the instructions don't mention whether the programs can be used with a 1581. Since the disks are not copy protected (a definite plus), it should be possible to copy them to a 1581 disk.

Mark Jordan, the author of the programs, invites users to write to him directly if they have problems or questions. That's a plus that's not often found with software these days.

The programs are powerful, easy to use, and very reasonably priced. The BrainStorm disk, which includes BrainPower, is only $22 (US), which includes shipping and handling. It's well worth the investment for anyone who's serious about getting organized in their writing.

**Story Writer**

Also available from Country Roads is Story Writer, which costs $12.00 (including shipping). Story Writer is a simple program to use; so simple, in fact, that it doesn't even come with a printed manual. There are instructions on the disk that you can print out, if you wish, but the program itself is pretty much self explanatory. The instruction file can be loaded into BrainPower, or any word processor or text display program that accepts SEQ files.

The first thing you'll see is a screen full of coloured boxes, each marked with an essential element of a story - setting, plot, protagonist, antagonist, conflict, and climax. You can choose any of them at any time by pressing its appropriate number.

When you choose a particular element, instructions for what should go in the box appear in a window at the bottom of the screen. Below the instructions is a scrolling "split screen" area into which you type. Simple text editing commands are available. After you press Return, the program gives you additional instructions on the kinds of things you should include in each area. For example, the "setting" box asks you for a specific setting for your story, and after you've provided that, it asks you for additional details to make your setting more authentic.

You can type in up to 100 ideas in each window. If you get confused, or want additional help, pressing the Help key brings up a series of help screens that give additional details on how the program works and what's expected in any particular window. In this way, the program guides you through the construction of a story, asking you for specific details and then giving you a chance to add more ideas of your own as you think of them. The more you supply, of course, the better your story will be. The program encourages you to think about what's happening, and gives good pointers on what should be included in which section.

You can edit the contents of any window at any time. The commands used to edit may take a bit of getting used to (you select an item to edit by pressing cursor-right, for example) but they work well. Each item to be edited is moved into the "split screen" area at the bottom of the screen. Once you're done, you can save your story to disk, or print it out. It won't appear in "story" form, of course, but rather as an outline which you can use to write the finished story. There is even a sample story on the disk which you can use as a guide.

The program is easy to use, and helps the writer think carefully about what should be included in a "good story." The best part of this program is that you can put your ideas down in any order in which they occur to you. You don't have to stick with any one element; you can jump back and forth as the Muse inspires you. And once you're done, you have all your work in order, ready to write the finished story. It's the BrainStorm concept applied to fiction, and it works remarkably well.

# C and Assembly: Clarifying the Link

*Not all static variables are created equal...*

**by Larry Gaynier**

*Larry Gaynier has more than 15 years of experience in software design, functional specification and documentation in a variety of mainframe environments. In this article, he builds on the description of C object files given by David Godshall in Volume 8, Issue 5 and gives explicit detail of how the "C Power/Power C" compiler handles static variables. Some of this material was also discussed by Adrian Pepper in Volume 9, Issue 1.*

I want to clarify some items presented in a recent *Transactor* article (*The Link Between C and Assembly* by David Godshall, *Transactor*, Volume 8, Issue 5). In my remarks, I assume "Power C" from Spinnaker is identical to "C Power" from Pro-Line. I have the "C Power" compilers for the C64 and the C128.

**Section 5: the static section**

The article described four sections that make up a C object file. However, there is a fifth section for static variables. The two NULLs, thought to terminate an object file, actually specify the length of the static section. In the example given, the length is zero, meaning there are no static variables.

An important characteristic of static variables is that they are always initialized to a known value at program start-up. The default initial value is zero unless an explicit value is included in the declaration for a static variable.

The static section of a C Power object file only contains static variables that rely on the default initialization to zero. Static variables that are explicitly declared with initial values, are handled in a different manner by the compiler. Unfortunately, this leads to inconsistent behavior with C Power programs as we shall see later.

Each entry in the static section consists of a null-terminated name string followed by a word giving the size in bytes.

Consider the following example:

```
int fun()
{
  char c;
  static int iarray[5];
  c = 1;
}
```

*Iarray* is declared to be a static array of five integers. When compiled using C64 C Power, the following object file is produced.

```
          hex dump of file fun.o

0000 1d 00 4c 4c 4c 85 fb a9   ..lll...
0008 01 a2 00 a0 00 20 20 20   ... .
0010 a2 01 a0 00 86 2b a9 01   .. ..+..
0018 a2 00 a0 00 4c 4c 4c 00   .. .lll.
0020 00 01 00 46 55 4e 00 01   ...fun..
0028 03 00 03 00 43 24 53 54   ....c$st
0030 41 52 54 00 00 00 00 00   art.....
0038 43 24 31 30 35 00 00 00   c$105...
0040 0b 00 43 24 31 30 36 00   ..c$106.
0048 00 00 1a 00 01 00 cd 29   ......M)
0050 49 41 52 52 41 59 00 05   iarray.H
0058 00
```

The sections in the object file can be easily identified based on their location relative to the beginning of the file.

```
Section    Location   Length   Contents
--------   --------   ------   --------------
code        $0000       $1D
relocate    $001F         0
global      $0021         1    fun
external    $002A         3    c$start,c$105,
                               c$106
```

The static variable section begins at location $004C showing a length of one. Next, comes the only static entry - *iarray*. Its size is declared to be ten bytes. The first two bytes in the name string are generated by the compiler. (I am not sure about their significance. They appear to be random identifiers assigned by the compiler).

As you can see, no data space has actually been allocated for *iarray*. This happens at run-time similar to automatic variables. During the linking process, the static variable entries are collected into a contiguous data area to be located immediately above the program in memory. When linking is complete, the executable program knows the starting address and size of the static data area. At run-time, the program performs a simple loop to zero each byte in the static data area, guaranteeing the static variables are initialized to zero.

**Inconsistent behavior**

If a static variable is explicitly declared with an initial value, appropriate memory is allocated and initialized in the code section after the JMP C$START but before the regular executable code. It does not appear in the static section. As a result, initialization happens once when the program is loaded. If the static variable changes value during execution, the new value is remembered and becomes the initial value for the next execution of the program, unless the program is reloaded. Consider the following example, which includes explicit initialization.

```
int fun1()
{
  char c;
  static int iarray[5] = {1,2,3,4,5};
  c = 1;
}
```

*Iarray* is declared to be a static array of five integers initialized to 1,2,3,4,5. When compiled using C64 C Power, the following object file is produced:

```
        hex dump of file fun1.o

0000 27 00 4c 4c 4c 01 00 02  '.lll...
0008 00 03 00 04 00 05 00 85  ........
0010 fb a9 01 a2 00 a0 00 20  ..... .
0018 20 20 a2 01 a0 00 86 2b    .. ..+
0020 a9 01 a2 00 a0 00 4c 4c  .... .ll
0028 4c 00 00 01 00 46 55 4e  l....fun
0030 31 00 01 0d 00 03 00 43  1......c
0038 24 53 54 41 52 54 00 00  $start..
0040 00 00 00 43 24 31 30 35  ...c$105
0048 00 00 00 15 00 43 24 31  .....c$1
0050 30 36 00 00 00 24 00 00  06...$..
0058 00                       .
```

The sections in the object file can be easily identified based on their location relative to the beginning of the file.

```
Section   Location  Length  Contents
--------  --------  ------  -------------
code       $0000      $27
relocate   $0029        0
global     $002B        1   fun1
external   $0035        3   c$start,c$105,
             c$106
static     $0057        0
```

Two things worth noting in this example are: the code section is larger and the static section is empty. This result occurred because *iarray* was allocated and initialized by the compiler beginning at location $0005.

My advice is to keep in mind the potential side effects when you explicitly initialize static variables as part of their declaration.

**Parameter passing**

I think the article contains an error in describing how parameters are passed during a function call. A character variable was claimed to be passed as one byte. Actually, a character variable is first converted to an integer when passed. This conversion is described in the book "The C Programming Language" by Kernighan and Ritchie. You may see other literature about the C language refer to the conversion as 'widening' or 'promoting.' Any actual arguments of type *float* are converted to *double* before the function call; any of type *char* or *short* are converted to *int*. The C Power compiler only widens character variables because the data type sizes are limited: *char* is one byte; *short*, *int*, *long*, *unsigned* and *pointer* are two bytes; *float* and *double* are five bytes. Consider the following example:

```
callfred()
{
  char age, *name;
  float weight;
  int height;
  fred(age,name,weight,height);
}
```

This function calls the function FRED that was described in the article. When compiled using C64 C Power, the following object file is produced.

```
        hex dump of file callfred.o

0000 48 00 4c 4c 4c 85 fb a9  h.lll...
0008 05 a2 05 a0 00 20 20 20  ....
0010 a9 00 20 20 20 a6 2b a0  ..    .+
0018 00 8e 3c 03 8c 3d 03 a6  ..<..=..
0020 2c a4 2d 8e 3e 03 8c 3f  ,-.>..?
0028 03 a9 04 a2 00 a0 00 20  ......
0030 20 20 a6 2e a4 2f 8e 45    ../.e
0038 03 8c 46 03 a9 0b 20 6c  ..f... .
0040 6c a9 05 a2 05 a0 00 4c  ......l
```

```
0048 4c 4c 00 00 01 00 43 41   ll....ca
0050 4c 4c 46 52 45 44 00 01   llfred..
0058 03 00 06 00 43 24 53 54   ....c$st
0060 41 52 54 00 00 00 00 00   art.....
0068 43 24 31 30 35 00 00 00   c$105...
0070 0b 00 43 24 31 31 30 32   ..c$1102
0078 00 00 00 10 00 43 24 31   .....c$1
0080 31 33 38 00 00 00 2d 00   138...-.
0088 46 52 45 44 00 00 00 3c   fred...<
0090 00 43 24 31 30 36 00 00   .c$106..
0098 00 45 00 00 00            .e...
```

On disassembly, it will be seen that the following 6502 code is produced:

```
Code Size:    72(D) / 48(X)

  0 relocation entries
  ;--------------------
  1 external definitions
  callfred        R  0003
  6 external references.

      *= $0000

  4c 4c 4c / 1800    jmp c$start
  ;--------------------
  callfred

      85 fb 00 / 0003    sta $fb
      a9 05 00 / 0005    lda #$05
      a2 05 00 / 0007    ldx #$05
      a0 00 00 / 0009    ldy #$00
      20 20 20 / 000b    jsr c$105
      a9 00 00 / 000e    lda #$00
      20 20 20 / 0010    jsr c$1102
      a6 2b 00 / 0013    ldx $2b
      a0 00 00 / 0015    ldy #$00
      8e 3c 03 / 0017    stx $033c
      8c 3d 03 / 001a    sty $033d
      a6 2c 00 / 001d    ldx $2c
      a4 2d 00 / 001f    ldy $2d
      8e 3e 03 / 0021    stx $033e
      8c 3f 03 / 0024    sty $033f
      a9 04 00 / 0027    lda #$04
      a2 00 00 / 0029    ldx #$00
      a0 00 00 / 002b    ldy #$00
      20 20 20 / 002d    jsr c$1138
      a6 2e 00 / 0030    ldx $2e
      a4 2f 00 / 0032    ldy $2f
      8e 45 03 / 0034    stx $0345
      8c 46 03 / 0037    sty $0346
      a9 0b 00 / 003a    lda #$0b
      20 6c 6c / 003c    jsr fred
      a9 05 00 / 003f    lda #$05
      a2 05 00 / 0041    ldx #$05
      a0 00 00 / 0043    ldy #$00
      4c 4c 4c / 0045    jmp c$106
```

As the example shows, the accumulator is loaded with eleven just before the JSR to FRED. These eleven bytes of parameters are passed to FRED in memory starting at $033c:

$033c - Age (two bytes, zero high byte)
$033e - Name (two bytes, pointer)
$0340 - Weight (five bytes, FP representation)
$0345 - Height (two bytes)

Once inside the function FRED, the upper byte of Age will be loaded to zero page storage but it will never be used. Widening of parameters has been generally recognized as an inefficiency of the C language. The new ANSI C standard. when adopted, will add function prototyping to the C language which will make parameter widening unnecessary. I am curious to see if the "C Power" compilers will be updated to match the ANSI C standard.

**Wrapup**

Overall, the C Power 128 compiler exhibits identical behavior as the C64 version except that parameters are passed in memory starting at $0400 in bank 1 and different zero page locations are used during function execution.

I hope you find this information useful to better understand the C Power compilers and to avoid some pitfalls.

# Care and Feeding of the C256

## *Fattening your C64*

by Paul Bosacki

It really wasn't that long ago that 64K was a lot of memory. But in five short years, we've gone from measuring 'enough' memory in kilobytes to megabytes. An Amiga comes with 512K, but apparently needs at least a meg to really show. We owners of the (dare I say it) venerable Commodore 64 have 64K and really little opportunity to expand beyond that other than the 1764 REU. But what if I told you that there is a way to expand the C64 to 256K, that you can do it yourself, and (best yet) I can promise you compatibility with GEOS? And maybe, just maybe, it can all happen for under $100.00. Interested? Read on.

This promised memory expansion comes in two parts. The first is a circuit whose construction is, I believe, not beyond the abilities of the novice. The second is the direct replacement of the memory chips with 41256's (a 256K bit DRAM). There is some interface wiring between the 64 and the mod board.

As you are aware, the 64K memory capacity of the C64 is dictated by the width of the address bus: sixteen bits. I have found an easy way to add two pseudo-address lines. These two new lines allow access to four banks of 64K each. When each bank is mapped in, the standard configuration of the C64 is unchanged. If all ROMs were mapped in, then they still are. Little has changed, except that there is now 256K rather than 64K resident in the machine. By the way, Bank 0 is always brought into context on power-up. As well, a certain amount of Bank 0 is always available. This is necessay for two very important reasons. Although much of memory can be changed at will without consequence, there are certain locations that the Operating System and the MPU need to have available at all times; namely, system vectors and the stack. Consequently, these locations are never mapped out.

### The modification

Three switches and two bits from the MPU I/O port are used to configure the additional memory. Briefly, P3 and P4 (the cassette write line and cassette sense line) provide the two pseudo-address lines needed to access the additional memory. By reconfiguring P4 to output and writing one of four two-bit codes to location $01, one of four banks of 64K can be mapped in. This technique must certainly ring a bell, for it is by the same method that the ROMs are mapped either in or out.

Of the three switches, one allows the machine to be significantly reconfigured. It allows A10 to be considered when decoding common ram. If considered, common ram (CRAM) is from $0-$03FF. Otherwise, CRAM exists to $07FF. This option sets the default screen matrix within CRAM. The result is that, on a bank switch, the screen remains unchanged and 'common' to each bank. Another switch allows the AEC line from the VIC Chip to generate a CRAM call. If enabled, a low on AEC causes Bank 0 to be mapped in (CRAM is always Bank 0). If disabled, AEC has no effect. This option is particularly useful with CRAM set to $03FF and AEC disabled. Each bank then has its own screen matrix which can be used in the usual sense (i.e. your typing appears on the screen). If AEC is enabled, the screen is 'protected.' It's visible, but the contents cannot be directly modified from the keyboard. Through these options any number of screens are made available without stealing Bank 0 RAM.

The third and final switch disables the bank select circuitry. This is done quite simply by driving the *STROBE pin of the 'LS157 high. This forces a low on output Y1 bringing in Bank 0. This capability is needed with software that is lazy when writing to the MPU I/O port. A bank switch might inadvertently occur. Further, software which is protected through the use of 'keys' that fit into the cassette port will not load properly without disabling the expansion memory. And of course, because we are using the cassette sense and write lines as our two pseudo-address lines, the datassette is incompatibile with this expansion project.

### Circuit theory

Briefly, an 'LS245 bus transceiver buffers A10-A15 to a pair of 'LS32 OR gates acting as address decoders. However, A10 is first AND'ed with a qualifying signal generated by SW1. If the qualifying signal is high (SW1 is open), A10 is reflected at the output of the gate and passed back to the OR gates. If low, A10 is inhibited, and a low is generated regardless of the state of A10.

The OR gates decode the address lines in this fashion: if any line should go high, then a high is generated at the final output. Otherwise, a low is generated indicating an attempt to access memory below $07FF (or $03FF if A10 is considered in the decoding). This signal is labled *CRAM (Common RAM) on the schematic. *CRAM is then AND'ed with *VID (AEC relabeled). *VID goes low only when the VIC chip updates the display. If *VID is enabled, then both signals must be high in order for the output of the 'LS08 AND gate to go high. When *VID is disabled through opening SW2, *VID is forced high by a 4.7K pull-up resistor. In such a situation, *CRAM is reflected at the output of the AND gate. This output is the BEN (bank enable) signal.

The BEN signal is really the heart of this expansion. P3 and *P4 (P4 is inverted earlier) are each presented to one input of an AND gate; BEN serves as a qualifer at the other input. When BEN is high, P3 and *P4 are present on the output of their respective gates allowing a selected two-bit code to be sent to pin 1 of the 41256 DRAM's. If BEN is low, P3 and *P4 are inhibited and a low is present at the outputs forcing a default to Bank 0. These two outputs are LA16 and LA17.

The two pseudo-address lines are then passed to a 'LS157 multiplexer. The *SELA line of the multiplexer is controlled by the same signal that drives the 'LS257 multiplexors on the system board. That is, the *CAS signal generated by the VIC chip. The two pseudo-addresses are multiplexed onto pin 1 of the 41256's at the same time as the rest of the address bus.

That's pretty much it. Dependent upon the BEN signal, which is enabled only when both *CRAM and AEC are high, four banks of 64K are available. It is worth noting that *VID goes low only on a video access, which itself disables the MPU, and that *CRAM only goes low when the MPU tries to access memory below $0800 or $0400 (depending upon whether A10 is considered).

**Installation**

The installation of this modification is absolutely a two step process. First, disconnect everything: printers, disk drives, joysticks and the power supply. Next, void your warranty by disassembling your C64. Remove the top RF shield, gently disconnect the keyboard, then remove the screws that hold the system board to the bottom half of the case. Now, remove the bottom RF shield. In some cases, it is held in place by the same screws and simply drops off. However, mine was soldered in place. Desoldering braid did the job for me.

Now life gets interesting. Locate the eight 4164 DRAM's on the system board. They're usually in the lower left corner. On my board they were labeled U9-U12 and U21-U24. Turn your board over and carefully note their position. Now comes the fun part: remove them. I used a combination of solder braid and a vacuum desolder. Make certain that each hole is as free of solder as possible. Each chip should then pry free fairly easily. If you want to save the chips remember to keep your time at each pin to a minimum. The heat generated by a soldering iron will quickly ruin a memory chip.

Once you've removed all eight, install 16-pin sockets and carefully solder them to the system board. Be certain not to leave solder bridges between pins. That's a sure headache later on. Once installed, use a fine gauge wire (I used wire wrap) to link together pin 1 of each socket. For now, connect pin 1 of the last socket to a convenient ground. Install the 41256's. They are extremely static sensitive so ground yourself first to discharge any static that may have built up. Now, reassemble your computer (at least set it back in its case). Reconnect the power supply and your monitor. Hold your breath and turn your machine on.

If you get the usual power on message, everything's fine for now. If you didn't, try reseating the 41256's. Are any upside down? Did a pin slip outside a socket? Carefully reinspect your soldering and try again. If your display flips (i.e., random display that changes in a random fashion), then one chip or more isn't seated properly. Check everything. Most likely, however, you will get the usual power up message.

With the DRAM's in place and working properly, the next step is construction of the mod board. I used point to point soldering, although wire wrap works just fine. If you can etch your own board that's great. For the board itself, I used one from Radio Shack with solder-ringed holes (cat. no. 276-158). Remembering to leave room for ground plane and +5V plane, install the sockets according to the wiring diagram and solder a couple of pins on each socket to hold them in place. Then run the ground and +5V planes. For the planes, try 'stitching' stripped 22 gauge through every fifth or sixth hole and apply a touch of solder. Make the appropriate +5V and ground connections, and install the capacitors at this time. Now, follow the wiring diagram and carefully complete the job. The wiring diagram shows the board from the component side. Remember that pin 1 of the 'LS245 is at the top *left* corner on the component side, and top *right* corner on the solder side.

Just a hint: after I had installed the sockets and power planes, I trimmed the board leaving a 1/4" border all around.

With the board finished, its time to hook it up. But first find a place for it. I have a 64C and placed mine between the VIC chip and the RF modulator. Don't fix it in place yet. Using ribbon cable five-conductor wide, follow the diagram of the Expansion Port on the following page (seen from the solder side) and solder one lead to each address line. I found that ribbon cable from Radio Shack was of a fine enough gauge that I could ease the Expansion Port lead to one side and insert the wire. Again watch out for solder bridges. The address bus will crash, and your computer will lock up with no display.

Determining just how long the cable will have to be is up to you, but keep it as short as possible. Solder the other end of the cable to your mod board, again each lead to one input of the 'LS245 socket.

Now we have to do a little hunting. Locate pin 1 of either 'LS257 (U13 or U25) on the component side of the system board (look to the right of the 41256's). This is the *CAS signal. Follow the trace away from the chip until you find a silver solder dot. This is a pass-through jumper to the other side of the board. Heat the solder and insert a single strand of ribbon cable about 10" long.

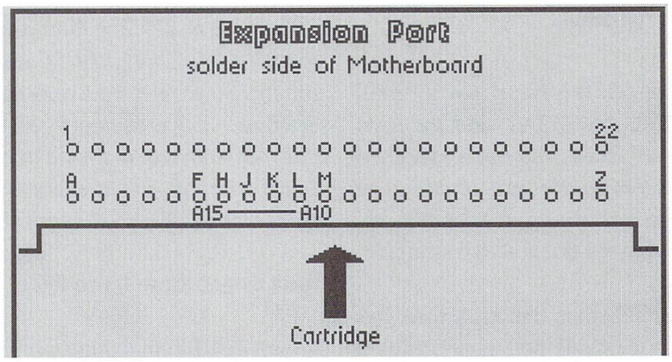



*Diagram of the C64's Expansion Port*

Now locate pin 16 of the VIC chip (U19). This is the AEC signal. Either locate a pass-through jumper as before, or from the solder side, heat the pin, ease it aside and insert another single strand of ribbon cable. Now make the appropriate connects to the mod board. AEC goes to SW1. *CAS goes to pin 1 of the 'LS157 multiplexer socket.

Look at the diagram of the Datassette Port. P3 and P4 are available on leads E5 and F6 respectively. P4 connects to pin 1 of the 'LS04 socket; P3 connects to pin 13 of the 'LS08 socket. I also took my power supply from the port. Use a heavier gauge wire to make the connections. Now connect the switches. I found a DIP array works well. Use enough ribbon cable so that you'll be able to pass the DIP switch out the cassette port. This way it'll be easier to get at it when you want to change the configuration of the mem board. Use shrink wrap to insulate the leads of the switch so that casual use doesn't result in two leads shorting to each other.

Don't yet install the chips. Reassemble as much of your computer as necessary and power up. Get the usual message? Great!

If you didn't, recheck your wiring, especially the connections to the expansion port and 'LS245 socket. Since none of the chips have yet been installed, your problem must lie somewhere in the interface wiring. So check it all over and find the fault.

If you have a logic probe, check the A inputs of the 'LS245 socket for high/low pulses. As well, pin 1 should show high and pin 19, low. If missing, check your wiring and the solder joints. Now, test P4 at pin 1 of the 'LS04 socket. Is it high? Do the same for P3 at pin 13 of the 'LS08 socket. It should be low. Check for pulse conditions where AEC and *CAS come to the mod board. Check each ground and +5V connection for the corresponding low or high. If everything checks out, turn off your computer and install the chips ensuring correct orientation and placement.

(Note: all of the above tests can be performed with a voltmeter. High/low pulses will show about 2.4v, low below 0.8V, and high above 3.5V.)

Power up and the usual message should appear. If it doesn't, recheck the above. Also look for pulses on the outputs of the OR and AND gates. Check the wiring of the mod board itself; this is most likely where the problem lies. If the usual message did appear, then turn off your computer and disconnect pin 1 of the 41256's from the ground and connect it to pin 4 of the 'LS157. Power up and you should be in business. Four banks of 64K each are just waiting for you. Reassemble your computer and promise yourself that you'll never take it apart again.

**Getting acquainted**

Try this from the direct mode with AEC disabled and A10 considered: **poke 1,63**. Your screen filled up with garbage, right? That's because the VIC chip now 'sees' Bank 1 RAM. Clear the screen and enter **list**. Again you probably got garbage or a syntax error. So, enter **sys58303** and then **new**. List again; nothing right? Now enter a short program:

```
10 for j=1 to 1000: next
```

Now enter **poke 1,55**. Something happened. The power-up screen (or a part of it) is back, and the cursor is about half way down the screen. Now list your program. Not there? Go back to Bank 1 and **list**. It's there, isn't it. Neat stuff, eh?

Now reconfigure bit 4 of the MPU data direction register to output with **poke0, peek(0) or 16**. Now, try going to each of the other banks. For the default power-on configuration, the values are:

| | |
|---|---|
| bank 0: | 55 |
| bank 1: | 63 |
| bank 2: | 39 |
| bank 3: | 47 |

Have fun with it. Your machine now contains 256K of memory! If you want, enter and run the Memory Test program supplied with this article. It will simply and quickly test the four banks of memory. Also get to know the table of configurations so you won't be too shocked when AEC is enabled, A10 not considered, and you switch banks.

### GEOS V 1.3 and 256K

At the beginning of this article, I promised compatiblity with GEOS. Using *Configure256*, you can configure a 1541 RAM disk to run under GEOS. The RAM disk uses the bank switched RAM we've just installed. If you've ever tired of what seemed like continuous disk access when running an application, life just got better. Loading an application from the RAM disk takes a second or two; overlays take no time at all. If a copy of the DESKTOP is also on the RAM disk, the DESKTOP comes back up almost instantly.

When you first run *Configure256* for the DESKTOP, a small block transfer routine is moved down to $02A7, and the RAM drive code at $9C80 is modified to access the bank switched RAM. Also, the number of drives on the system is stored at $C013. This serves to indicate to the program whether it's been run before and, if it has, to bypasses the above routines.

Next a check is performed to test whether the RAM disk has been formatted. If so, the program exits. If not, it performs a format, then exits. Device 9 is the default RAM drive. On a one-drive system, a new disk icon appears below the other with the name BRAM 1541. On two-drive systems, the icon appears in the same place, but because the DESKTOP can handle only two drives, your second 1541 cannot be accessed.

On subsequent calls to *Configure256*, a menu is displayed offering the following options:

a) **format:** just what it implies
b) **Rdrive A:** move RAM drive to device 8 (only enabled when there are two disk drives on the system)
c) **Rdrive B:** as above
d) **flip:** exchange RAM drive and 1541 at that device number. On a system with two drives, this allows you to have three drives availalble (though only two at a time).
e) **1541:** remove RAM drive from system.
f) **quit**

The menu appears immediately to the right of the DESKTOP menu.

There are some points to note while using this patch. The first is that A10 must be set to 'considered' and for esthetic reasons, AEC must be enabled (video display drawn from Bank 0). The RAM disk table uses RAM from $0400 to $FFFF in Banks 1-3. If A10 is not considered, the Bank 0 RAM at $0400 will be overwritten, leading to a *System Error Near...* message.

Next, this program, and in fact, this expansion module is incompatible with the Calculator DA. The DA writes to $01 in a very sloppy manner causing an inadvertent bank switch. If you use the Calculator, switch off the mod board. It can be done on the fly, so you can switch it back on afterward.

GEOS must be run from Bank 0. GEOS writes to $01, and knows nothing about our Ram Expander. If you *do* manage to boot, and run *Configure256*, you will get an error message. Shut down and reboot from Bank 0. Otherwise, you're only asking for trouble.

Lastly, there is absolutely no such thing as free memory under GEOS. This program has been tested with applications like *geoWrite 2.0*, *geoCalc*, *geoPublisher*, the *geoProgrammer* package, the *Desk Pack* and found to be compatible. However, I cannot guarrantee compatibility with applications not listed.

There are also some limitations when running this program under GEOS. *Configure256* does not support the shadow drive, nor the Stash, Fetch, Swap and Verify routines. If you run the *Configure256* program from your boot disk, it will indicate a 1541 RAM Drive, but RAM expansion will be "none." Don't attempt to configure a shadow drive. If you do, rerun *Configure256* and click on "Rdrive B."

### How to get there from here

A question I kept asking myself throughout this project: where do I go from here? It been 11 months since I modified my C64 to run with 256K, and I'm still coming up with new ideas and uses for my expansion memory. Here are a few of the more interesting ones. About 1.8K of the Kernal is devoted to cassette operations; how about replacing that code with stash and fetch routines? That way precious CRAM is not cluttered with transfer routines. Or how about a modified CHRGET routine to fetch BASIC text from another bank?

You've got nearly 196K of expansion RAM. How many graphic screens is that? Easy animation?

Or something more down to earth: a few days ago, I was working on a problem that required a sector editor, an ML monitor and a couple of other support utilities. If I hadn't had the expansion RAM, I would have been loading one, using it, loading another, using it and so ad infinitum. As it was, I loaded each into its own bank and switched banks when I needed to.

Here's something on the hardware front. On the C128, you can declare 32K of CRAM through the MMU. How about adding another AND gate and switch (along the lines of the A10 inhibit circuitry), and declaring more CRAM. Or add A9 to the address decoding. With A9 considered, CRAM would only go to $01FF. Each bank could have its own BASIC and OS vectors. Four computers in one!

I leave it up to your capable hands and imagination.

### Table of configurations

A) The four banks:

| | |
|---|---|
| Bank 0: | 55 |
| Bank 1: | 63 |
| Bank 2: | 39 |
| Bank 3: | 47 |

Bit 4 of the MPU DDR must be set to output (=1). By default, bit 4 is set to input and pulled high by a resistor. For this reason, bit 4 is inverted at the mod board.

B) Three switches offer 'hardset' configurations that can be switched on the fly:
i) **A10 consider:** considered when open
ii) **AEC enabled:** enabled when open
iii) **Disable Mod:** disabled when open.

a) A10 considered/AEC enabled: CRAM $02-$03FF. Video display data drawn from Bank 0, however OS 'sees' current bank. Therefore, screen seems to freeze until Bank 0 reset. Type blindly: **poke1,55**.

b) A10 considered/AEC disabled: CRAM $02-$03FF. Video display is drawn from the current bank. OS also sees this bank, and updates accordingly.

c) A10 not considered/AEC enabled: CRAM $02-$07FF. Default screen matrix now falls within CRAM. All banks share a common screen.

d) A10 not considered/AEC disabled: CRAM $02-$07FF. Reverse of configuration (a). Video display data drawn from current bank, however OS 'sees' Bank 0. Type blindly: **poke1,55**.

**Notes:**

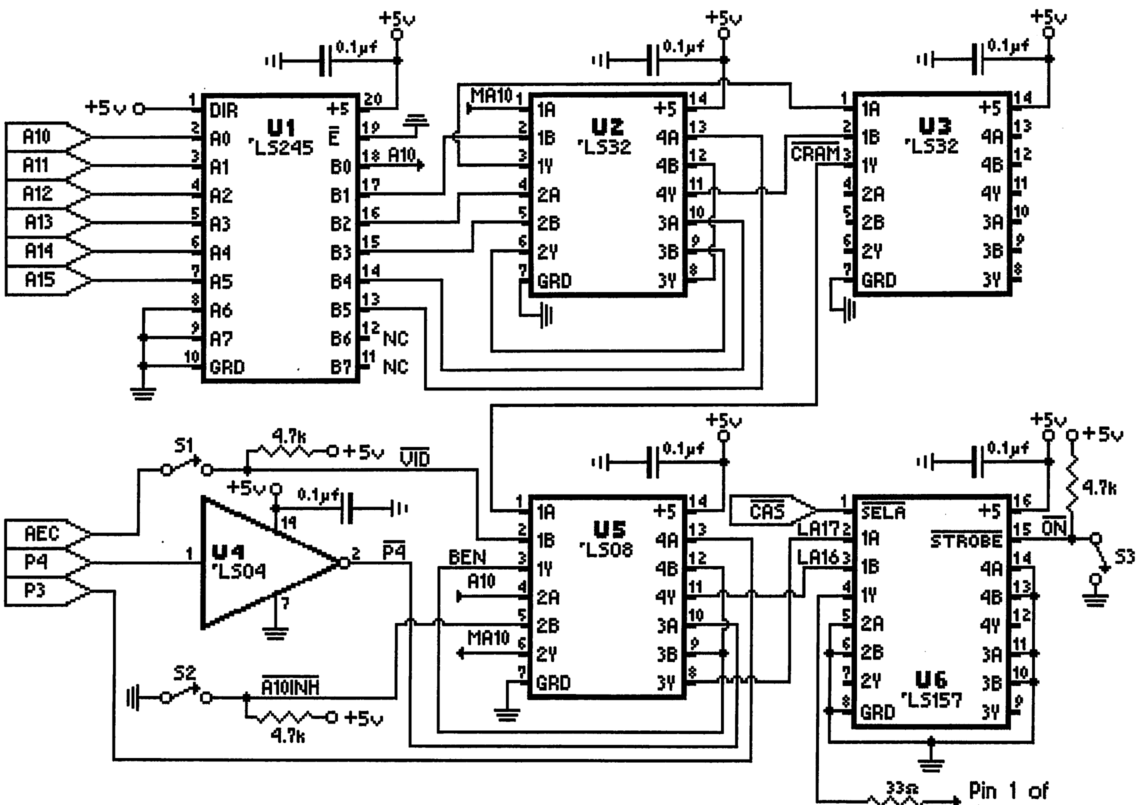
When A10 is considered and AEC is enabled the cursor is 'lost' when Bank 0 is switched out. If AEC is then disabled, the cursor is found in the current bank. Why?

Bank 0 is only brought into context when:
1) address falls below least decoded line (LDL), or
2) AEC goes low indicating VIC DMA.

If LDL is A10 and screen memory is at the default location, the OS acts on the current bank whose screen matrix lies above CRAM. In this sense, the cursor isn't lost. The OS is happily updating the cursor and the video matrix. However, since it knows nothing about the expansion ram, its updating the current bank. The reverse is true when A10 is not considered and AEC is disabled.

**Disclaimer:** While we have every confidence that Mr. Bosacki's project functions as it should, we would like to make it clear that it was not undertaken here at *Transactor* and we were therefore unable to test the software. Consequently, we would like to request that you refer any problems or suggestions to Mr. Bosacki directly, rather than sending them to *Transactor*. You may be interested to know that Mr. Bosacki is now using a *one megabyte* C64. He has 512K installed inside that machine and a 512K REU. We would be remiss if we failed to inform you that a project such as the one discussed will certainly void your warranty. That being said, please address your letters to:

Paul Bosacki
37-1443 Huron St.
London, ON, Canada
N5V 2E6



MemEx Daughter Board

```
;****************************************
;*          Config256.hdr               *
;*                                      *
;* This file contains the header block *
;* definition for Config256.            *
;****************************************

.if Pass1
  .include  Config256.sym
.endif

;Here is the header. The Config256.lnk file will
;instruct the linker to attach it to application.

.header            ;start of header section

.word  0           ;first 2 bytes are always zero
.byte  3           ;width in bytes
.byte  21          ;and height in scanlines of:

.byte  $80|USR     ;C= file type, with bit 7 set.
.byte  APPLICATION ;GEOS file type
.byte  SEQUENTIAL  ;GEOS file structure type
.word  $400        ;start address (where to load)
.word  $3ff        ;usually end address, but only
                   ;needed for desk accessories.
.word  $400        ;init address (where to JMP)
.byte  "RamDriver   V1.1",0,0,0,$00
                   ;permanent filename: 12 chars,
                   ;then 4 character ver. number,
                   ;then 3 zeroes,
                   ;then 40/80 column flag.
.byte  "Paul J. Bosacki     ",0
                   ;twenty character author name

                   ;end of header section which
                   ;is checked for accuracy
.block 160-117     ;skip 43 bytes...
.byte  "Configures banked RAM to act"
.byte  "as RAMdrive as under GEOS",0
.endh

;******************************************
;*          Config256.lnk                 *
;*                                        *
;* Here are the link file directives for *
;* Config256.                             *
;******************************************

.output configure256      ;application name
.header Config256.hd.rel  ;header definitions
.seq                      ;sequential file type
.psect  $0400             ;loads in at $400
 Config256.mn.rel         ;link this file

;*********************************************
;*             Config256.Sym                 *
;*                                           *
;* These are the GEOS equates for Config256 *
;*********************************************

;*** Geos OS_TAB Routines Used ***

CmpFString      ==      $c268
DoDlgBox        ==      $c256
DoIcons         ==      $c15a
DoMenu          ==      $c151
DoPreviousMenu  ==      $c190
EnterDeskTop    ==      $c22c
GetBlock        ==      $c1e4
MoveData        ==      $c17e
NewDisk         ==      $c1e1
PutBlock        ==      $c1e7
ReDoMenu        ==      $c193
SetDevice       ==      $c2b0

;*** Misc. Equates ***

CPU_DATA      ==     $0001
CPU_DDR       ==     $0000
curDevice     ==     $00ba
curDrive      ==     $8489
diskBlkBuf    ==     $8000
dispBufferOn  ==     $002a
numDrives     ==     $848d

;*** Geos Constants ***

APPLICATION  =    6
BOLDON       =    24
DEF_DB_POS   =    $80
DBSYSOPVEC   =    14
DBTXTSTR     =    11
DIR_TRACK    =    18
HORIZONTAL   =    0
MENU_ACTION  =    0
NULL         =    0
SEQUENTIAL   =    0
ST_WR_FORE   =    $80
TXT_LN_X     =    16
TXT_LN_2_Y   =    32
TXT_LN_3_Y   =    48
USR          =    4
VERTICAL     =    $80

;*** Geos Pseudo-Register Definitions ***

r0           ==    $0002
r0H          ==    $0003
r0L          ==    $0002
r1           ==    $0004
r1H          ==    $0005
r1L          ==    $0004
r2           ==    $0006
r2H          ==    $0007
r2L          ==    $0006
r3           ==    $0008
r3H          ==    $0009
r3L          ==    $0008
r4           ==    $000A
r4H          ==    $000B
r4L          ==    $000A

;*********************************************
;*          RamDisk Driver (GEOS)            *
;*                                           *
;* This program creates a RAMdisk            *
;* driver for GEOS. Runs as an application. *
;* Leaves a transfer routine and patch.      *
;*********************************************
;
.if Pass1
  .include Config256.sym
  .include geosMac
.endif
;
.zsect $02          ;zpage begins
                    ;at r0
source:             .block 2
destination:        .block 2
tlength:            .block 2
Sbank:              .block 1
Dbank:              .block 1
;
; CPU_DATA: This is the Bank Control Reg.
; Four banks of 64K are available.
; The register is laid out like this:
;
; bit 0: ROM select
; bit 1: ROM select
; bit 2: ROM select
; bit 3: bank select (LA16)    see schematic
; bit 4: bank select (LA17)     "      "
; bit 5: cassette motor
; bit 6: n/a
; bit 7: n/a
;
C_numDrives   ==  $c013
BankMove      ==  $02a7
.psect
;
Init: LoadB dispBufferOn, #ST_WR_FORE
      lda C_numDrives       ;program called
      beq 11$               ;first time?
      jmp FromDesk
11$:  lda CPU_DATA          ;do initalization
      and #%00010000        ;are we in bank0?
      cmp #%00010000        ;if not, put up
      beq 10$               ;error message and
      LoadW r0,#NotB0_Tab ;exit to DeskTop.
      jsr DoDlgBox
      jmp Do_quit
10$:  lda curDevice
      sta Copy_curDevice
      lda numDrives
      sta C_numDrives
      jsr Install_Drive      ;install code patch
      LoadW r4, #diskBlkBuf  ;and move transfer
      LoadB r1L,#DIR_TRACK   ;routine to BankMove
      LoadB r1H, #0
      jsr GetBlock
      LoadW r0, #diskBlkBuf+144   ;offset to
      LoadW r1, #headerTitle+144  ;diskname
      ldy #$02
      ldx #$04
      lda #$10
      jsr CmpFString  ;has ramdisk already been
      beq 12$         ;formated? If so bypass
      jsr Format_Rdri ;format routine.
12$:  jmp Do_quit
;
FromDesk:
      lda curDevice        ;save current device
      sta Copy_curDevice
      LoadW r0, #Menu_Tab ;put up selection menu
      lda #0               ;pointer is placed
      jsr DoMenu           ;on first item
      rts
      ;
BX_TOP     =0
BX_HEIGHT  =$0c
BX_LEFT    =0    ;last two used as word values
BX_WIDTH   =$89  ;when declaring menu dimensions
;
Menu_Tab: .byte BX_TOP
          .byte BX_TOP+BX_HEIGHT
          .word BX_LEFT+BX_WIDTH
          .word BX_LEFT+BX_WIDTH+35
          .byte 1|HORIZONTAL
          .word DiskText
          .byte VERTICAL
          .word DiskSubMenu
;
```

```
DiskSubMenu:
        .byte BX_TOP+13
        .byte BX_TOP+(14*7)+1
        .word BX_LEFT+BX_WIDTH
        .word BX_LEFT+BX_WIDTH+75
        .byte 6|VERTICAL
        .word Txt_format
        .byte MENU_ACTION
        .word Do_format
        .word Txt_dra
        .byte MENU_ACTION
        .wrd Do_dra
        .word Txt_drb
        .byte MENU_ACTION
        .word Do_drb
        .word Txt_flip
        .byte MENU_ACTION
        .word Do_flip
        .word Txt_noram
        .byte MENU_ACTION
        .word Do_noram
        .word Txt_quit
        .byte MENU_ACTION
        .word Do_quit
;
DiskText:   .byte "RDrive", NULL
Txt_format: .byte "format", NULL
Txt_dra:    .byte "RamDri A", NULL
Txt_drb:    .byte "RamDri B", NULL
Txt_flip:   .byte "flip", NULL
Txt_noram:  .byte "1541", NULL
Txt_quit:   .byte "quit", NULL
;
;
NotB0_Tab:.byte DEF_DB_POS|1
        .byte DBTXTSTR
        .byte TXT_LN_X
        .byte TXT_LN_2_Y
        .word N_bk0_1
        .byte DBTXTSTR
        .byte TXT_LN_X
        .byte TXT_LN_3_Y
        .word N_bk0_2
        .byte DBSYSOPVEC, 0
;
N_bk0_1:
     .byte BOLDON, "This module can only be", 0
N_bk0_2:
     .byte BOLDON, "installed from BANK 0", 0
;
RDevice:  .byte #9  ;default ramdisk=device 9
Copy_curDevice:     .block 1
;
Do_format:
     jsr ReDoMenu
     lda #2
     sta numDrives
     jsr Format_Rdri
     rts
;
Do_dra:
     jsr ReDoMenu
     lda C_numDrives ;if there is only one 1541
     cmp #1          ;on the system then this
     bne 1$          ;option does not allow a
     rts             ;ramdrive as device #8
1$:  ldy #8
     sty RDevice
     bne Set_Device
Do_drb:
     jsr ReDoMenu
     ldy #9
     sty RDevice
Set_Device:
     lda #2
     sta numDrives      ;update # of drives
     lda #$81
     sta $8486,y
     cpy #8
     bne 5$
     iny
10$: and #%00000001     ;make drive type 1541
     sta $8486,y
     tya
     jsr SetDevice ;pass device # in acc.
                   ;Set_Device make that device
     jsr NewDisk   ;and updates drive varibles
     rts           ;initializes new disk
5$:  dey
     bra 10$
;
Do_flip:  jsr DoPreviousMenu  ;roll up menu
          ldy #8
          lda $8486,y
          pha
          iny
          lda $8486,y
          dey
          sta $8486,y
          iny
          pla
          sta $8486,y
          bmi 10$
          tya
          jsr SetDevice
          jsr NewDisk
          jmp Do_quit
10$:      dey
          tya
          jsr SetDevice
          jsr NewDisk
          jmp Do_quit
;
Do_noram: jsr ReDoMenu      ;simply put, restores
          lda #1            ;1541's to system.
          ldy #8            ;Checks C_numDrives
          sta $8486,y       ;for the number of
          lda C_numDrives ;1541's.
          cmp #1
          bne 10$
          lda #0
          ldy #9
          sta $8486,y
          lda #1
          sta numDrives
          jmp Do_quit
10$:      lda #1
          ldy #9
          sta $8486,y
          jmp Do_quit
;
Do_quit:  lda numDrives
          cmp #1
          bne 1$
          lda #8
          sta Copy_curDevice
1$:       lda Copy_curDevice ;restore curDrive
          jsr SetDevice      ;on entry to
          jmp EnterDeskTop   ;application.
;
; Geos disk routines read a block at a time.
; Because of this, our transfer routine requires
; only enough code to transfer 256 bytes.
;
s_BankMove:
     PushB CPU_DATA
     PushB CPU_DDR
     ora #%00010000      ;set CASS Sense to
     sta CPU_DDR         ;output. Restored at
     ldy #0              ;exit from routine
     ldx Dbank           ;from stack.
20$: lda Sbank
     sta CPU_DATA
     lda (source),y
     stx CPU_DATA
     sta (destination),y
     iny
     bne 20$
;
     PopB CPU_DDR
     PopB CPU_DATA
     rts
;
;
e_BankMove:
;
BankLen = e_BankMove-s_BankMove
     ; *** Patch GEOS Routines and
     ; transfer move block routine***
;
RamCode  == $9c80
;
Install_Drive:
     LoadW r0,#s_BankCode ;install RAM driver
     LoadW r1,#RamCode    ;code. This code then
     LoadW r2,#CodeLen    ;overwrites the
     jsr MoveData         ;existing GEOS routine
;
     LoadW r0, #s_BankMove  ;move transfer
     LoadW r1, #BankMove    ;routine to
     LoadW r2, #BankLen     ;BankMove.
     jsr MoveData
     rts
;
s_BankCode:
     ldx #0   ;dummy routine for the
     lda #0   ;verify RAM call
     rts
     nop
     nop
     nop
;
; following code replaces the GEOS RAMex driver.
;
     ldx #1             ;fetch
     bne 10$
     ldx #0             ;stash
10$: PushW r0
     PushW r1
     PushW r3
     ldy destination
;
; GetBlock and PutBlock routines pass
; the track value in r1L, and sector in r1H.
;
     dey
     lda RDTab,y        ;get RAM track value
     clc
     adc destination+1  ;add in sector as offset
     sta destination+1
     lda #%00111000     ;value for Bank1
     cpy #11            ;if track<11 then Bank1
     bcc 40$
```

```
        lda #%00100000      ;value for Bank2
        cpy #23             ;if track<23 then Bank2
        bcc 40$             ;else Bank3
        lda #%00101000      ;value for Bank3
40$:    sta Dbank
        ldy #%00110000
        sty Sbank
        ldy #0
        sty destination
        ldy r4H
        sty source+1
        ldy r4L
        sty source
        cpx #1      ;if Fetch then flip source and
                    ;destination as well as source
        bne 60$     ;and destination banks.
        PushB Dbank
        MoveW destination, source
        MoveW r4, destination
        MoveB Sbank, Dbank
        PopB Sbank
60$:    jsr BankMove  ;jump to block transfer
        PopW r3       ;routine. restore
        PopW r1       ;pseudo-registers
        PopW r0
        ldx #0        ;disk errors passed in x.
        lda #0        ;0=no error.
        rts

;RamDiskTab is a table of RAM page values used
;by the RAMex driver in order to determine the
;location of each track and sector. It works like
;this: each value represents Track XX, Sector 0,
;where XX is a validtrack. The sector number is
;then used as an offset. For example, Track 1,
;Sector 8 =4+8=12, or, Bank1, RAM page 12.

RamDiskTab:
        .byte $04,$1a,$30,$46,$5c,$72,$88,$9e
        .byte $b4,$ca,$e0,$04,$1a,$30,$46,$5c
        .byte $72,$88,$9c,$b0,$c4,$d8,$ec,$04
        .byte $18,$2b,$3e,$51,$64,$77,$8a,$9c
        .byte $ae,$c0,$d2,$e4
;
e_BankCode:
;
RDTab     = RamCode+(RamDiskTab-s_BankCode)
CodeLen   = e_BankCode-s_BankCode;

Format_Rdri:
        ldy RDevice     ;get RAMdrive #.
        lda #$81        ;high bit set=RAM device,
                        ;therefore $82=1571
        sta $8486,y     ;under geos.
        sty curDrive
        LoadW r4, #header       ;set up for PutBlock
        LoadB r1L,#DIR_TRACK    ;write header out to
        LoadB r1H, #0           ;18/0
        jsr PutBlock
        ldy #0
        tya
10$:    sta diskBlkBuf,y
        dey
        bne 10$
        lda #$ff
        sta diskBlkBuf+1
        LoadW r4, #diskBlkBuf
        LoadB r1L, #19          ;set up GEOS border
                                ;directory block &
        LoadB r1H, #08          ;write out to 19/8
        jsr PutBlock
        LoadW r4, #diskBlkBuf
        LoadB r1L, #DIR_TRACK  ;pass same block
        LoadB r1H, #1          ;to 18/1
        jsr PutBlock
        lda #0
        ldx #0                 ;no errors
        rts
;
header:
  .byte $12, $01, $41, $00  ;exactly what the
                            ;name implies the
  .byte $15, $ff, $ff, $1f  ;header block
  .byte $15, $ff, $ff, $1f  ;writen out by
  .byte $15, $ff, $ff, $1f  ;the PutBlock
  .byte $15, $ff, $ff, $1f  ;routine to 18/0
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $15, $ff, $ff, $1f
  .byte $11, $fc, $ff, $07
  .byte $12, $ff, $fe, $07
  .byte $13, $ff, $ff, $07
  .byte $13, $ff, $ff, $07
  .byte $13, $ff, $ff, $07
  .byte $13, $ff, $ff, $07
  .byte $13, $ff, $ff, $07
  .byte $12, $ff, $ff, $03
  .byte $12, $ff, $ff, $03
  .byte $12, $ff, $ff, $03
  .byte $12, $ff, $ff, $03
  .byte $12, $ff, $ff, $03
  .byte $12, $ff, $ff, $03
  .byte $11, $ff, $ff, $01
  .byte $11, $ff, $ff, $01
  .byte $11, $ff, $ff, $01
  .byte $11, $ff, $ff, $01
  .byte $11, $ff, $ff, $01
headerTitle:    .byte "BRam 1541"

      .byte 160,160,160,160,160
      .byte 160,160,160,160
      .byte $52, $44, $A0, $32
      .byte $41, $A0, $A0, $A0
      .byte $A0, $13, $08
      .byte "GEOS format V1.0"
      .byte 0,0,0,0,0,0,0,0,0,0,0,0
      .byte 0,0,0,0,0,0,0,0,0,0,0,0
      .byte 0,0,0,0,0,0,0,0,0,0,0,0
      .byte 0,0,0,0,0,0,0,0,0,0,0,0
      .byte 0,0,0,0,0,0,0,0,0,0,0,0
      .byte 0,0,0,0,0,0,0,0,0,0,0,0
;
ProgEnd:
.end

;*****************************************
;*          MemTest.lnk                  *
;*                                       *
;* Here are the link file directives for *
;* MemTest.                              *
;*****************************************

; The program MemTest.mn links to $033c: the
; cassette buffer. Start MEMTEST with "sys828."

.output   MEMTEST ;name of the output.
.cbm              ;the linked file will
                  ;be in the standard pre-Geos
.psect    $033c   ;format. Cassette buffer
                  ;always lies within CRAM.
    MemTest.rel   ;link this file

;*********************************************
;*               MemTest.mn                  *
;*                                           *
;* This program tests each bank of           *
;* memory, setting each bit=0 then each      *
;* bit=1. It runs from and returns to BASIC. *
;*********************************************

.if     Pass1
  .include      geosMac
.endif

;equates used:
a0          == $fb
a0L         == $fb
a0H         == $fc
CPU_DDR     == $00
CPU_DATA    == $01
CLRSCR      == $e544
STROUT      == $able
SCRTCH      == $a642

NULL        =  0

;This is not a comprehensive memory test, nor is
;it meant to be. Simply, each bit throughout an
;entire bank is set to one and then tested for
;this value. Then each bit is set to zero, and
;tested again. On failure the screen is cleared
;and an error message is generated. A10 must be
;set to "considered" for a thorough test. If AEC
;is disabled, then the screen will change each
;time a new phase of the test is entered
;providing a visual record of the test. The
;program runs in less than 30 secs., and then
;returns to BASIC.

Progstart: lda CPU_DATA      ;push configuration
;onto stack to be restored on exit from routine.
      pha
      lda CPU_DDR
      pha
      ora #%00010000      ;set bit 4 to output
      sta CPU_DDR
      lda #%00110000      ;all ram Bank0
      sta CPU_DATA
      jsr MemTest
      lda #%00111000      ;all ram Bank1
      sta CPU_DATA
      inc bankValue       ;ascii value
;used by errMess to indicate Bank in which
;fault occurred.
      jsr MemTest
      lda #%00100000      ;all ram Bank2
      sta CPU_DATA
      inc bankValue
      jsr MemTest
      lda #%00101000      ;all ram Bank3
      sta CPU_DATA
      inc bankValue
      jsr MemTest
```

```
        PopB CPU_DDR        ;restore configuration
        PopB CPU_DATA
        cli                 ;clear interrupt flag
                            ;set by MemTest routine
        jsr CLRSCR
        lda #0
        jsr SCRTCH
        rts

testvalue: .byte $ff

MemTest:  lda #255          ;first set each bit of
                            ;each location to one.
          sta testvalue
          sei
MemLoop:  lda #4            ;start address to test
;in this case $0400. This is the lowest CRAM can
;be set allowing each bank to be fully tested.
        sta a0H
        lda #0
        sta a0L
        tay
        lda testvalue
1$:     sta (a0), y         ;write test value out
        iny
        bne 1$
        inc a0H
        bne 1$
        lda #4
        sta a0H
        lda #0
        sta a0L
        tay
2$:     lda (a0), y
        cmp testvalue
        bne 3$
        iny
        bne 2$
        inc a0H
        bne 2$
        lda testvalue       ;if testvalue=0 then
                            ;set up for 2nd pass.
        bne 4$
        rts

4$:     inc testvalue       ;now set each bit=0
        jmp MemLoop

3$:     PushB CPU_DATA      ;save configuration
        ora #7
        sta CPU_DATA        ;and map in ROM's
        lda #[errMess
        ldy #]errMess
        jsr STROUT          ;use BASIC routine to
                            ;output error message.
        PopB CPU_DATA
        rts

errMess:  .byte 147,11," FAULT FOUND IN"
          .byte " MEMORY: BANK"

bankValue:.byte 48           ;ascii "0"
          .byte 11,11,13,NULL
```

*The following macros are from the file 'geosMac' supplied with geoProgrammer.*

```
.macro LoadB dest,value
        lda #value
        sta dest
.endm
.macro LoadW dest,value
        lda #](value)
        sta dest+1
        lda #[(value)
        sta dest+0
.endm
.macro MoveB source,dest
        lda source
        sta dest
.endm
.macro MoveW source,dest
        lda source+1
        sta dest+1
        lda source+0
        sta dest+0
.endm
.macro add source
        clc
        adc source
.endm
.macro AddB source,dest
        clc
        lda source
        adc dest
        sta dest
.endm
.macro AddW source,dest
        lda source
        clc
        adc dest+0
        sta dest+0
        lda source+1
        adc dest+1
        sta dest+1
.endm
.macro AddVB value,dest
        lda dest
        clc
        adc #value
        sta dest
.endm
.macro AddVW value,dest
        clc
        lda #[(value)
        adc dest+0
        sta dest+0
    .if (value >= 0) && (value <= 255)
        bcc noInc
        inc dest+1
      noInc:
    .else
        lda #](value)
        adc dest+1
        sta dest+1
    .endif
.endm
.macro sub source
        sec
        sbc source
.endm
.macro SubB source,dest
        sec
        lda dest
        sbc source
        sta dest
.endm
.macro SubW source,dest
        lda dest+0
        sec
        sbc source+0
        sta dest+0
        lda dest+1
        sbc source+1
        sta dest+1
.endm
.macro CmpB source,dest
        lda source
        cmp dest
.endm
.macro CmpBI source,immed
        lda source
        cmp #immed
.endm
.macro CmpW source,dest
        lda source+1
        cmp dest+1
        bne done
        lda source+0
        cmp dest+0
    done:
.endm
.macro CmpWI source,immed
        lda source+1
        cmp #](immed)
        bne done
        lda source+0
        cmp #[(immed)
    done:
.endm
.macro PushB source
        lda source
        pha
.endm
.macro PushW source
        lda source+1
        pha
        lda source+0
        pha
.endm
.macro PopB dest
        pla
        sta dest
.endm
.macro PopW dest
        pla
        sta dest+0
        pla
        sta dest+1
.endm
.macro bra addr
        clv
        bvc addr
.endm
.macro smb bitNumber,dest
        pha
        lda #(1 << bitNumber)
        ora dest
        sta dest
        pla
.endm
.macro smbf bitNumber,dest
        lda #(1 << bitNumber)
        ora dest
        sta dest
.endm
.macro rmb bitNumber,dest
        pha
        lda #[~(1 << bitNumber)
        and dest
        sta dest
        pla
.endm
```

```
.macro rmbf bitNumber,dest
        lda #[~(1 << bitNumber)
        and dest
        sta dest
.endm
.macro bbs bitNumber,source,addr
        php
        pha
        lda source
        and #(1 << bitNumber)
        beq nobranch
        pla
        plp
        bra addr
    nobranch:
        pla
        plp
.endm
.macro bbsf bitNumber,source,addr
    .if (bitNumber = 7)
        bit source
        bmi addr
    .elif (bitNumber = 6)
        bit source
        bvs addr
    .else
        lda source
        and #(1 << bitNumber)
        bne addr
    .endif
.endm
.macro bbr bitNumber,source,addr
        php
        pha
        lda source
        and #(1 << bitNumber)
        bne nobranch
        pla
        plp
        bra addr
    nobranch:
        pla
        plp
.endm
.macro bbrf bitNumber,source,addr
    .if (bitNumber = 7)
        bit source
        bpl addr
    .elif (bitNumber = 6)
        bit source
        bvc addr
    .else
        lda source
        and #(1 << bitNumber)
        beq addr
    .endif
.endm
```

# NORTHWEST MUSIC CENTER INC.

539 N. Wolf Rd. — Wheeling, IL 60090
Hours— (Voice) Phone (312) 520-2540
Mon.-Thurs. 12:30-5:00, Sat. 12:00-4:00
(24 Hour Order Recorder)

# We want to be your Commodore shop!

## Do you feel like no one cares about you and your orphaned computer! NWM cares!! We still stock and sell most major B series and 8000 series programs.

*Parts available for some 8000, 9000 and B series models!*

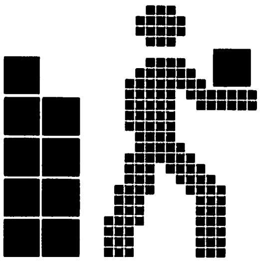
## Hacker's Corner

**One of a Kind • Surplus • Monthly Special • Closeouts**

Limited quantities to stock on hand

| | |
|---|---|
| 64 k ieee to parallel buffer | $199.00 |
| 4023p 100cps rehab | $99.00 |
| 64k ram exp 8032 | $110.00 |
| Smith Corona DM-200 | $179.95 |
| 6400, 8300p, Diablo 630 ribbons | $4.95 |
| 8023p and MPP 1361 ribbons | $5.50 |
| 9090 7.5 meg rehab | $495.00 |
| 4023p ribbons | $6.00 |
| Everex 2400 external modem | $245.00 |

## NWM's INVENTORY CONTROL SYSTEM*



- loads program modules in less than **8 seconds** (superbase 2) to main menus in **3 seconds** or less
- on screen **pop-up calculator** in transaction modules
- most data centered function use the **calculator keypad**
- versatile report features allow for **3 ways** to print the same report. User selects the fastest method
- built in sophisticated export program allows for **complete packing** of the database
- type ahead feature allowed
- you can display reports on screen
- access to superbase menu for user developed applications

| | |
|---|---|
| **B Version 1 8050** | **$39.95** |
| **B Version 2 8050** | **$39.95** |
| **C-128 Version 1 1571** | **$24.95** |
| **B-128 Version 1&2 8050** | **$44.95** |

*Requires use of superbase

## Commodore's Superpet 9000

only $199.00

*while supplies last*

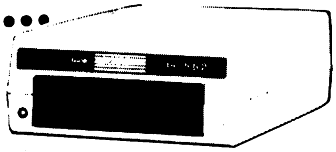
**With Five Interpretive Languages:**
**Cobol**
**Pascal**
**Basic**
**Fortran**
**Apl**

**Runs 8032 software.**
**Great for schools and students**

## 64K Memory Expansion for 8032 only $110 upgrades your 8032 to an 8096.

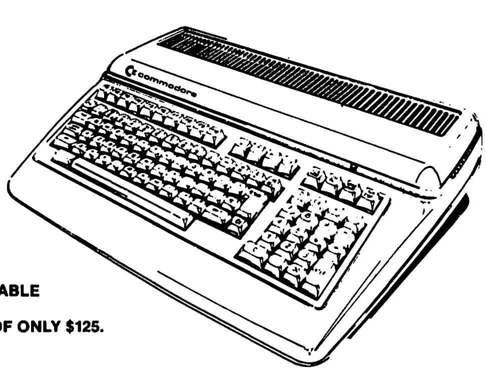
### COMMODORE 8000-9000 SOFTWARE & MISC.

| | |
|---|---|
| 9000 Superpet | $199.00 |
| 64K exp for 8032 | $110 |
| OZZ Database | $25 |
| BPI General Ledger | $25 |
| BPI Accts Payable | $25 |
| BPI Job Cost | $25 |
| BPI Accts Receivable | $25 |
| BPI Inventory | $25 |
| Superscript 8032 | $79 |
| Superbase 8096 | $79 |
| Legal Time Acc | $25 |
| Dow Jones Program | $25 |
| Info Designs 8032 Accounting System | $50 |
| Superoffice 8096 | $149 |
| Calc Result 8032 | $89 |



## B-128 $145 U.S.

**NEW 128K USER INSTALLABLE MEMORY EXPANSION! INTRODUCTORY PRICE OF ONLY $125.**



## SFD 1001 1 Megabyte
PRICED AT **$149.95** (US)
$125 with purchase of Superpet

SFD-1001 is the drive that you should consider when you need large amounts of data storage. It holds over 1 megabyte of data on its single floppy drive. Fast IEEE access for your C-64 or C-128. (C-64 and C-128 need an IEEE interface.) Why settle for slower drives with less storage capacity. This drive stores substantially more programs and data. Think how much money you can save on disk purchases. In fact, it stores almost 7 times more information than your standard drive. Bulletin board owners love them. And what an introductory price! At $169.95 these drives will sell fast, so don't wait. This drive has the identical format of a CBM 8250 drive, one of Commodores most durable floppy drives.

| | |
|---|---|
| MODEL | SFD-1001 |
| DRIVES | 1 |
| HEADS/DRIVE | 2 |
| STORAGE CAPACITY (Per Unit) | |
| Formatted | 1.06 Mb |
| MAXIMUM (Each Drive) | |
| Sequential File | 1.05 Mb |
| Relative File | 1.04 Mb |
| Disk System | |
| Buffer RAM (Bytes) | 4K |
| DISK FORMATS (Each Drive) | |
| Cylinders (Tracks) | (77) |
| Sector/Cylinder | — |
| Sector/Track | 23-29 |
| Bytes/Sector | 256 |
| Free Blocks | 4133 |
| TRANSFER RATES (Bytes/Sec) | |
| Internal | 40 Kb |
| IEEE-488 Bus | 1.2 Kb |
| ACCESS TIME (Milli-seconds) | |
| Track-to-track | * |
| Average track | ** |
| Average Latency | 100 |
| Speed (RPM) | 300 |

## SOFTWARE FOR THE B-128!!!

| | |
|---|---|
| Superbase | $19.95 |
| Superscript | $19.95 |
| Superoffice Integrated | |
| Superbase & Superscript | $49.95 |
| Calc Result | $89.95 |
| Word Result | $89.95 |
| Super Disk Doc | 24.95 |
| The Power of: Calc Result (Book) | $14.95 |

C.A.B.S. Accounting

| | |
|---|---|
| General Ledger | $ 9.95 |
| Accounts Receivable | $ 9.95 |
| Accounts Payable | $ 9.95 |
| Order Entry | $ 9.95 |
| Payroll | $ 9.95 |
| Buy all 5 for only $24.95 | |
| Superbase: The Book | $14.95 |
| Applied Calc Result (Book) | $14.95 |

## ORDER NOW WHILE STOCK LASTS!

Send or call your orders to: **Northwest Music Center, Inc. 539 N. Wolf Rd., Wheeling IL 60090. 312-520-2540** For prepaid orders add $25.50 for Superpet, $10.45 SFD 1001, $11.45 B-128, $10.45 4023p, $16.45 9090 and $5.45 64K memory expansion. For software add $3.50 for first and $2.00 for each additional book or program. Canadian shipping charges are double U.S. For C.O.D. orders add $2.20 per box shipped. All orders must be paid in U.S. funds. Include phone numbers with area codes. Do not use P.O. Box, only UPS shippable addresses. A 2 week hold will be imposed on all orders placed with a personal or business check. C.O.D. orders shipped in U.S. only and cash on delivery, no checks. 30 day warranty on all products from NWM, Inc. No manufacturer warranty. NWM reserves the right to limit quantities to stock on hand and adjust prices without notice!

**All prices quoted in US dollars.**

# News BRK

## Submitting News BRK Press Releases

If you have a press release you would like to submit for the News BRK column, make sure that the computer or device for which the product is intended is prominently noted. We receive hundreds of press releases for each issue and those whose intended readership is not clear must unfortunately go straight into the trash bin. We only print product releases which are in some way applicable to Commodore equipment. News of events such as computer shows should be received at least six months in advance. The News BRK column is compiled solely from press releases and is intended only to disseminate information; we have not necessarily tested the products.

**Turbo Master CPU Processor Accelerator for C64:** Schnedler Systems' Turbo Master CPU accelerator cartridge speeds up the operation of the Commodore 64. A replacement microprocessor clocked at 4.09 MHz provides four times faster processing speed, not merely a disk speed-up. BASIC, word processor scrolling, spreadsheets, assemblers, graphics and GEOS are all accelerated. In addition, 'turbo' disk routines are included in ROM for five times faster disk load and save, as well as a DOS 'wedge'. The plug-in cartridge features an onboard 65C02 processor, 64K RAM and a 32K EPROM.

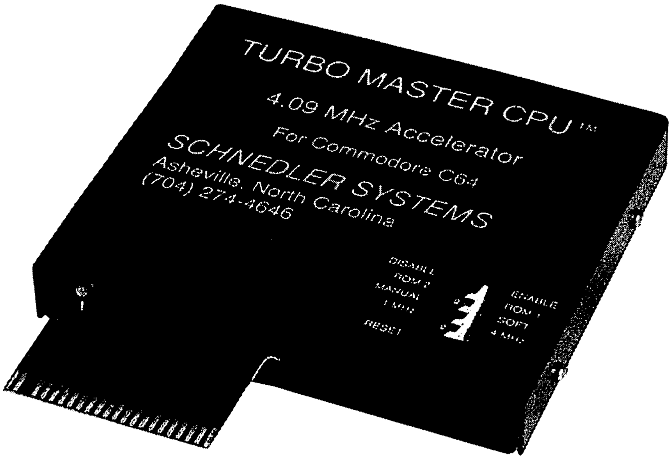
According to the manufacturer, Turbo Master CPU is compatible with nearly all C64 software. This includes programs written in BASIC, ML programs, GEOS applications, programs that use bank switching to access RAM under ROM, programs that move screen memory, and bit-mapped graphics screens. The main limitation is that the few ML programs that use 'illegal' or undocumented 6502 op-codes will not run because Turbo Master CPU uses an enhanced 65C02 in which many previously undocumented op-codes are now officially assigned as useful instructions.

Turbo Master CPU is intended for regular C64 and 64C computers. It also works with the SX-64 portable. It is not intended for the C128, even in C64 mode. The 'turbo' disk routines are for a 1541 disk drive, or close compatible. The 'turbo' disk routines can easily be turned off if you are using a different drive, without affecting the faster processing speed of the computer. Cassette tape cannot be used with Turbo Master CPU. Turbo Master CPU can be used with Schnedler Systems data acquisition and control interface boards, but not with other cartridges. (However the possibility of an adapter to allow use in conjunction with a 1764 REU is being investigated.)

Turbo Master CPU is almost exactly the size of an REU. It has a metal case with a gold-plated circuit board edge connector extending about 1.5 inches.

In addition to the Rockwell R65C02P4 clocked at 4.09MHz, Turbo Master includes its own 64K of fast static RAM (120 ns) and a 32K EPROM. The on-chip port on the regular 6510 processor in the C64 is emulated with TTL logic for bank switching. There are 16 ICs in all. In fact, Turbo Master CPU is practically a complete 64K microcomputer, lacking only a keyboard and screen. During operation of Turbo Master CPU, the 6510 processor in the C64 is completely bypassed, while the VIC chip, SID chip, keyboard and serial bus port are all accessed by the cartridge and operate normally. The 6526 CIA timers in the C64 continue to be clocked at their usual 1.0225 MHz rate. Speed change between the standard 1.0225 MHz processor clock rate and the fast 4.09 MHz clock rate can be accomplished either manually using the switches, or by software. (The software speed switch is bit 7 of memory address $00.) Schnedler Systems also markets the MAE assembler, which recognizes the full 65C02 instruction set.

Included in the cartridge is a switch for selecting ROM 1 or ROM 2. ROM 1 has no tape routines, replacing them with the fast disk routines and a DOS wedge. ROM 1 is highly compatible but ROM 2 is nevertheless provided. The cassette tape routines are present, but are disabled. ROM 2 does not have any extra features compared to a standard CBM Kernal ROM, but you still have the benefit of four times faster processing speed.




Turbo Master CPU comes with a 24-page manual, plus a disk containing a few demo and utility programs, as well as text files for assembly language programmers documenting the enhanced 65C02 instruction set.

The introductory price of the Turbo Master CPU is $179 US, shipping prepaid to US addresses. Orders may be placed by telephone or mail. Visa and MasterCard accepted. Ten-day satisfaction or money back guarantee. Order from: Schnedler Systems, 25 Eastwood Rd., P.O. Box 5964, Asheville, NC, 28813. Telephone: (704) 274-4646. Shipment from stock within 24 hours via UPS.

**Dialogue 128:** Advanced Terminal Software for the C128 and C128D. Over two years of work has gone into making Dialogue 128 the ideal terminal, with features no other software can match. All commands are accessible via keyboard, help menus, or 1351 mouse. Colour graphics mode for both CBM and IBM type ANSI bulletin boards is supported.

Features include: VT52 and VT100 terminal emulation; 300, 1200, 2400 and variable baud rates; support of major modem types through individual modem support files; all operations in 2MHz fast mode.

Visually, the package offers: 40/80 column selectable display, optional 25 or 50 line display and extensive in-program drop-down help screens. Memory usage options include: full support of 1700 and 1750 REUs and C128 RAMDOS (installed at page 11); 64,000 character capture buffer (800/1600 lines); 8000 character separate review buffer; optional 512,000 character buffer with RAM expander (6400/12,800 lines); multiple buffer configurations (1 to 8 separate capture buffers).

Dialogue 128 also includes: a full-featured text editor; extensive autodial/redial capabilities; the ability to autodial multiple numbers and load and save multi-dial lists; a 30-entry phone directory with individual terminal parameters and four user-defined function keys for each number in directory; ten other "always active" macro keys; split screen conference mode; clock and alarm for both "real" time and on-line time; up/downloading with Punter C1 or Xmodem (CRC and checksum) protocols.

A powerful auto-execute script language allows the advanced user the option of creating logon scripts or even developing programs to enable unattended operation. This feature could be used to set up a mini BBS.

According to the manufacturer, loadable extension files vastly extend Dialogue's future capabilities. Extension files are included for RLE files (allows viewing of RLE graphics while online and storing them to disk) and for fully integrated character set editing. Dialogue 128 is not copy protected - use any drive or combination of drives. Files are accessible via a convenient file selection mode. Dialogue 128 uses burst mode for 1571 and 1581 disk drives and permits use of partitions when using 1581 drives. It will also let you send printer commands via the DOS wedge.

Sending in your registration card entitles you to access the Workable Concepts BBS (directly or via PunterNet) and the Workable Concepts newsletter.

Dialogue 128 was written by Gary Farmaner and is available from: Workable Concepts Inc., 281 St. Germain Avenue, Toronto, ON, Canada, M5M 1W4. Price is $59.95 (Cdn) or $49.95 (US).

**QDisk Non-Volatile C128 CP/M RAM disk:** Brown Boxes Inc. of Bedford, MA and Herne Data Systems Ltd., of Toronto, ON are pleased to announce the release of QDisk version 2.0. QDisk is a device driver for the Quick Brown Box that allows it to be used as a non-volatile RAM disk in C128 CP/M mode. QDisk is totally application-transparent and can be used with all standard CP/M software such as PIP, WordStar and dBase.

The Quick Brown Box is a battery-backed CMOS static RAM cartridge for use with the C64 and C128 computers. It is available in 16K, 32K and 64K byte sizes for $69, $99 and $129 respectively (prices in US dollars; add $3 shipping and handling, and 5% sales tax in MA). The internal lithium battery retains the contents of the RAM for up to ten years, even when the cartridge is unplugged from the computer. It is supplied with RAM disk software for use on a C64 and C128 (in native mode). With the introduction of QDisk, the speed and flexibility of a non-volatile RAM disk is now available for C128 CP/M mode also.

In addition to being able to use the entire 64K version as a single CP/M drive, QDisk allows partitioning of the 64K Quick Brown Box into two 32K areas, either one of which can be used for C64 or C128 native mode applications, or both of which can be used as separate CP/M drives. Once the driver is installed, the Quick Brown Box can be accessed as a normal CP/M disk drive. However, unlike the standard C128 CP/M RAM disk using the 1700/1750 REUs, QDisk does not lose its contents when the computer is turned off. Programs and data files remain stored until needed and can be recalled in an instant.

QDisk is available for $9.95 (US) or $10.95 (Cdn.) plus $2.00 shipping and handling from: Herne Data Systems Ltd., P.O. Box 714, Station 'C', Toronto, ON, M6J 3S1. Phone: (416) 535-9335. For more information about the Quick Brown Box, contact: Brown Boxes Inc., 26 Concord Road, Bedford, MA, 01730. Phone: (617) 275-0090 or (617) 862-3675.

**B-128 Hardware Enhancements**

The low-profile B-128 is being supported by a line of hardware enhancement products from Anderson Communications Engineering.

The **B-1024** is a 1MB memory expansion board that implements banks 0-14 with socketed 256K dynamic RAM. This increase, for example, allows 8 documents with SuperScript II; 28,560 input values with Calc Result; and 14 8032s with the multi-tasking 8432 Emulator software package. The board is fully assembled and tested. It plugs internally onto pin fields in the low profile B-128 and installs in minutes. The original configuration of a stock B-128 can be restored at any time by simply removing the board; this is a non-destructive upgrade. In addition to the megabyte of memory there are pin fields for future I/O, a RAM/ROM socket to implement $0800-$1FFF in BANK 15, documentation for installation, parts layout, schematic diagram, and a machine language memory test utility on 8050 disk. Three memory density options are available. The board can be fully populated with user-purchased 41256-15 DRAM. Pricing: B-1024 with 1024K installed is $699, with 512K installed is $499, with 0K installed is $289.

The **24K RAM/ROM Cartridge** adds another 24K of memory to your B-128/CBM-256 system unit in BANK 15 from $2000-$7FFF. It comes complete in a plastic case with 24K of SRAM, is assembled and tested, and is ready to be plugged into the cartridge port. This cartridge is used with many software packages available from the Chicago B-128 Users Group (CBUG, 4102 N. Odell, Norridge, IL 60634) such as Scott's BMON, JCL Workshop, Harrison's Assembler, Jarvis/Springer Serial Bus Software, and Liz Deal's Keytrix to name a few. Price: $84.95

Serial Bus peripherals can now be connected to the B-128! The **Serial Bus Interface** is a full featured hardware interface for the B-128 implementing the Commodore serial bus with the functions of controller, listener/talker, slow bus, fast bus, attention acknowledge, power-on serial bus reset, and manual serial bus reset. It comes in a rugged plastic case and connects to the user port via a ribbon cable. This interface operates with the Jarvis/Springer Serial Bus Software Package available from CBUG. Price: $59.95

The **RAM/ROM Socket** allows implementation of the 6K memory area below the cartridge port from $0800-$1FFF in Bank 15. It is a small circuit board with connector and 28-pin socket that will receive an 8K SRAM or ROM to customize your applications. Price: $24.95

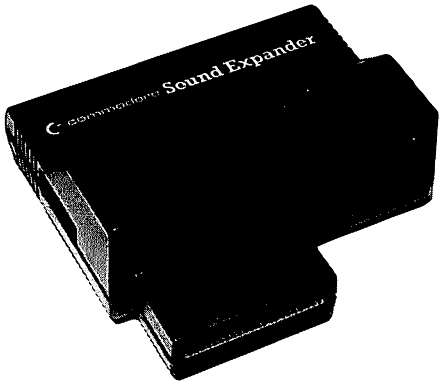
A copy of the latest ad can be obtained by sending an SASE to Anderson Communications Engineering, 2560 Glass Road NE, Cedar Rapids, IA 52402. Terms: free shipping in USA, US funds, Iowa add 4%, allow 6-8 weeks.

**SFX Sound Expander:** This expansion module for the C64/C128 is based on a custom LSI chip that provides a programmable nine voice FM synthesizer on a single chip. The manufacturer reports a great improvement in sound quality and variety over that of the Commodore SID chip. The basic software package that comes with the Sound Expander is a menu-driven program that includes a large selection of voices (more can be added with the optional FM Composer and Sound Editor program), keyboard split, chorus, transposition, one-finger chords, a rhythm machine, a riff machine and more. Also included with the Sound Expander is a Programmer's Reference Guide that gives all the programming details.

The SFX full-size keyboard combines with the SFX Sound Expander to create a "truly professional musical instrument", the manufacturer says. The keys on the five-octave keyboard are of standard piano key dimensions. The SFX keyboard overlay is a small two-octave keyboard that fits on top of the computer.

FM Composer and Sound Editor: This software is compatible with the SFX Sound Expander module. It is a MIDI-compatible nine-channel sequencer developed in cooperation with England's Reading University; it allows you to enter, edit and play back any piece of music that can be written in standard music notation. Among the features supported are: crescendo and decrescendo, loudness settings for individual notes (ppp to fff), repeats (da coda, dal Segno, start etc.), transposition, tempo settings, performance voice (may be changed for every note if desired, choosing from up to 64 voices in any composition), key, detuning (for chorus effects), copy and move options.

The Sound Editor program provides the tools to create, edit, store, and recall a wide variety of sounds for use with the Sound Expander. It also includes a random sound generator that lets the computer generate sounds on its own, a MIDI-compatible synthesizer mode and a drum machine.




SFX Sound Expander is $180.00 (US); SFX Full-Size Keyboard is $145.50; SFX Keyboard Overlay is $14.00; SFX Composer and Sound Editor is $45.00; SFX Sound Sampler is $89.00. Available from: Fearn & Music, 519 W. Taylor #114, Santa Maria, CA, 93454. Call: (800) 447-3434. In CA, call (805) 925-6682.

**Zoom! for the C64:** Discovery Software International, Inc. has announced the release of ZOOM!, the company's second arcade-style game for the C64/C128. The game, described as easy-to-learn and non-violent, features a character named Zoomer, who is chased by a gang of reckless enemies through an outer space land called Zoomland. Zoomer's mission is to dart through Zoomland, capturing territories and collecting points.

According to the manufacturer, Zoom! is designed to appeal to video game enthusiasts at all skill levels, and challenges players' reflexes and strategic thinking abilities. The game has 50 levels of play for one or two players. Suggested retail price of Zoom! is $29.95 (US). The game carries a 30-day unconditional money-back guarantee.

# The Potpourri Disk

### Help!

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

### Loan Helper

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

### Keyboard

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

### Filedump

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

### Anagrams

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

### Life

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

### War Balloons

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

### Von Googol

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

### News

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

### Wrd

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

### Quiz

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

### AHA! Lander

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

### Bag the Elves

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

### Blackjack

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

### File Compare

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

### Ghoul Dogs

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

### Octagons

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

### Backstreets

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

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

## The ENTIRE POTPOURRI COLLECTION JUST $17.95 US!!

See Order Card at Center

THE SIXTH ANNUAL

# THE WORLD OF COMMODORE

**December 1 – 4, 1988**
**International Centre**
**Toronto**

Incredible bargains.
Amazing new hardware and software.
The excitement.
It all returns to Toronto for the sixth great year: Canada's annual computer extravaganza sponsored by Commodore Business Machines.

The 1987 show set world records – 42,000 showgoers made it the largest, best-attended Commodore show ever.

Commodore will be at the 1988 show with a giant display area for its exciting computers, accessories and software.

Thousands of brands will be shown and sold by other manufacturers, distributors and retailers.

Top experts in home, business and educational computing will lead seminars and demonstrations which are free with admission for all show visitors.

The Amiga, the C-64, the C-128, the Commodore PC line, all the major peripherals, programs, accessories – many more to be announced – they're all at the Sixth World of Commodore.

Get in on the excitement.

**Exhibitors, contact:**
**The Hunter Group**
**(416) 595-5906**
**Fax (416) 595-5093**

Produced in association with Commodore Business Machines