Canada $4.25
USA $3.50

Transactor

THE MAGAZINE FOR COMMODORE PROGRAMMERS APRIL 1989: VOLUME 9, ISSUE 4

- Loadermaker - Take the pain out of GEOS program development
- Fattening the C128 - RAM expansion alternatives
- Customizing C128 CP/M - Patches for CPM+.SYS
- How random is RND? An analysis of the C64 and C128 RND routines
- Inside GEOS 128 - Information you won't find anywhere else
- The C64 Power C Shell - Making it work with a RAM expansion unit
- Turning off write/verify - Modifying 1571 vectors
- Converting 1541 disks to 1571 format
- An introduction to GEOS files - Using the high-level disk routines
- **Product Reviews**: Macro Set 1 from Xytec, X-10 Powerhouse Computer Interface, SFX Sound Expander, *What's Really Inside The C64?* reviewed by Jim Butterfield
- **Plus** Regular columns by Todd Heimarck and Joel Rubin, *Bits,* and more

ThunderBear by Wayne Schmidt

UTILITIES UNLIMITED, Inc.

12305 N.E. 152nd Street
Brush Prairie, Washington 98606

If you wish to place your order by phone, please call **206-254-6530**. Add $3.00 shipping & handling; $3.00 COD on all orders. Visa, M/C accepted. Dealer Inquiries Invited.

WORLD'S BIGGEST PROVIDER OF C64/128 UTILITIES

Software Submissions Invited

We are looking for HACKER STUFF: print utilities, parameters, telecommunications, and the unusual.

We now have over 1,000 parameters in stock!

OVER 5000 UNITS SOLD!!!

Unlike our competitors, we at Utilities Unlimited, Inc. have been concentrating all our efforts in bringing the newest technology. The result of that effort is **SuperCard**. It is far **superior to all the copy utilities out there including:**

Ramboard/Renegade, Datel Burst Nibbler, 21Second, Ultrabyte, and any other backup utility on the market. So don't be led astray. **We will give you your money back** if they can back up more of the latest software, will they??? In a word "NO! ALL SALES ARE FINAL!!!" That is their response if you want to return RAMBO.

If you happen to see the ads on RAMBOard (original name huh), they claim to be cheaper. Well, that's partially true, but as is usual, mostly false. First you need to buy their board, then you need to spend another $34.95 for software to run their board. **That makes the cost of Rambo/Renegade to be at least $69.90. But then they claim you can use our software (what does that say about their software?). Well now, that may be just a bit of a white lie as well, while it's true that early, less reliable versions work with THEIR thing, the new more reliable versions of SuperCard software is specifically designed not to work with their RAMBO. For those people that have found out that the RAMBO and Renegade software package are quite inferior to SuperCard we offer the following suggestion. Send in your RAMBO and $24.95 and WE'LL SEND YOU THE REAL THING — SuperCard.** Needless to say you need a pair of hip boots to walk through their claim that they are the best. By the way, their software that backs up an unprotected disk in 50 seconds, well, it doesn't even use the RAMBO to work. **I suppose if you had a choice of an OLDSMOBILE or a Corvette with no engine, you would still pick the Oldsmobile.**

SuperCard 1541/1541c	**$49.95**	2 drive version	**$79.90**
SuperCard 1541-II	**$59.95**	2 drive version	**$99.90**
SuperCard 1571	**$59.95**	2 drive version	**$99.90**

SuperCard 1541-II version will work with most compatible drives.

These prices include software. You don't need to steal anyone else's software to make it work.

SUPER PARAMETERS 500 Pack #1 and #2

500 Pack #1 - $24.95 has the vintage parameters on it that no one else has. This pack comes in a 5-disk set.
500 Pack #2 - $29.95 has all the most current parameters on it. And put together as only Utilities Unltd. can. All Super Parameter Packs are completely menu driven, fast and reliable. Included on both 500 Packs is our state-of-the-art 64/128 Super Nibbler at no extra charge.

SUPER PARAMETERS 1000 Pack #1

Utilities Unltd. has done it again!! We have consolidated and lowered the prices on the most popular parameters on the market . . . Super-Parameters, now you can get 1000 parameters and our 64/128 nibbler package for just **$39.95!!!** This is a complete 10 disk set, that includes every parameter we have produced.

PARAMETERS CONSTRUCTION SET

The company that has **The Most Parameters** is about to do something **Unbelievable.** We are giving you more of our secrets. Using this **Very Easy** program, it will not only **Read, Compare** and **Write Parameters for You;** it will also **Customize** the disk with your name. It will impress you as well as your friends. The **"Parameter Construction Set"** is like nothing you've ever seen. In fact you can even **Read Parameters** that you may have already written; then by using your construction set rewrite it with your new **Customized Menu. $24.95**

LOCK PICK - THE BOOKS - for the C64 and C128

Lock Pik 64/128 was put together by our crack team, as a tool for those who have a desire to see the **Internal Workings** of a parameter. The books give you **Step-By-Step Instructions** on breaking protection for backup of 100 popular program titles. Uses **Hesmon** and **Superedit.** Instructions are so clear and precise that anyone can use it.

• OUR BOOK TWO IS NOW AVAILABLE •

BOOK 1: Includes Hesmon and a disk with many utilities such as: **KERNAL SAVE, I/O SAVE, DISK LOG FILE** and lots more, all with instructions on disk. Along-time favorite.

BOOK 2: 100 NEW EXAMPLES, Hesmon on disk and cartridge plus more utilities to include: **A General Overview on How to Make Parameters** and a **Disk Scanner. $19.95 each** OR BUY BOTH FOR **ONLY $29.95**
Now with FREE Hesmon Cartridge.

THE 128 SUPERCHIP - A, B or C (another first)

A — There is an empty socket inside your 128 just waiting for our **Super Chip** to give you 32K worth of great **Built-in Utilities,** all at just the **Touch of a Finger.** You get built-in features: **File Copier, Nibbler, Track & Sector Editor, Screen Dump,** and even a 300/1200 baud **Terminal Program** that's 1650, 1670 and Hayes compatible. Best of all, it doesn't use up any memory. To use, simply touch a function key, and it responds to your command.

B — **HAS SUPER 81 UTILITIES,** a complete utility package for the 1581. Copy whole disks from 1541 or 1571 format to 1581. Many options include 1581 disk editor, drive monitor, Ram writer and will also perform many CP/M & MS-DOS utility functions.

C — **"C" IS FOR COMBO** and that's what you get. A super combination of both chips A and B in one chip, switchable at a great savings to you. **All Chips Include 100 Parameters FREE!**
Chips A or B: **$29.95 ea.** Chip C: **$44.95 ea.**

SUPER GRAPHICS 1000 PACK

That's right! Over 1000 graphics in a 10-disk set for only **$29.95.** There are graphics for virtually everything in this package. These graphics work with Print Shop and Print Master.

ADULT GAME & GRAPHICS DATA DISKS

GAME: A very unusual game to be played by a very **Open Minded** adult. It includes a **Casino** and **House of Ill Repute.** Please, you **Must be 18** to order **Either One.**
DATA ★: This Popular disk works with **Print Shop** and **Print Master.**
Now Version 1 + 2 . . . **$24.95 ea.**

NEW! SUPER CARTRIDGE EXPLODE! V4.1 w/COLOR DUMP $44.95

Introducing the **World's First Color Screen Dump** in a cartridge. Explode! V4.1 will now **Support Directly** from the screen. FULL COLOR PRINTING for the **Rainbow Star NX-100** and also the **Okidata 10 & 20** printers.

The **Most Powerful Disk Drive** and **Printer Cartridge** produced for the COMMODORE USER. **Super Friendly** with the features most asked for.

- **SUPER FAST** built-in single drive 8 or 9 FILE COPY, copy files of up to 235 BLOCKS in length, in less than 13 seconds!
- **SUPER SCREEN CAPTURE.** Capture and Convert **Any Screen** to KOALA or DOODLE.
- **SUPER FAST FORMAT** (8 SEC'S) - plus FULL D.O.S. WEDGE w/standard format!
- **SUPER FASTLOAD** and **SAVE** (50k in 9 SEC'S) works with all **C-64 or C-128's No Matter What Vintage!** And with most after market drives EXCEPT the 1581, M.S.D. 1 or 2.
- **SUPER PRINTER FEATURES** allows ANY DOT MATRIX PRINTER even 1526/802 to print HI-RES SCREENS (using 16 shade GRAY SCALE). **Any Printer or Interface Combination** can be used with SUPER EXPLODE! V4.1 or V3.0.
- **NEW and IMPROVED CONVERT** feature allows anybody to convert (even TEXT) **Screens into DOODLE or KOALA Type Pictures w/Full Color!**
- **SUPER FAST SAVE** of EXPLODE! SCREENS as KOALA or DOODLE FILES w/COLOR.
- **SUPER FAST LOADING** with **Color Re-Display of DOODLE or KOALA** files.
- **SUPER FAST LOAD or SAVE** can be TURNED OFF or ON without AFFECTING the REST of SUPER EXPLODE'S FEATURES. The rest of Explode V4.1 is still active.
- **SUPER EASY LOADING and RUNNING** of ALL PROGRAMS from the DISK DIRECTORY.
- **SUPER BUILT-IN TWO-WAY SEQ. or PRG. file READER** using the DISK DIRECTORY.
- **NEVER TYPE A FILE NAME AGAIN** when you use SUPER EXPLODE'S unique LOADERS.
- **CAPTURE** 40 COLUMN C or D-128 SCREENS! (with optional DISABLE SWITCH). Add $5.

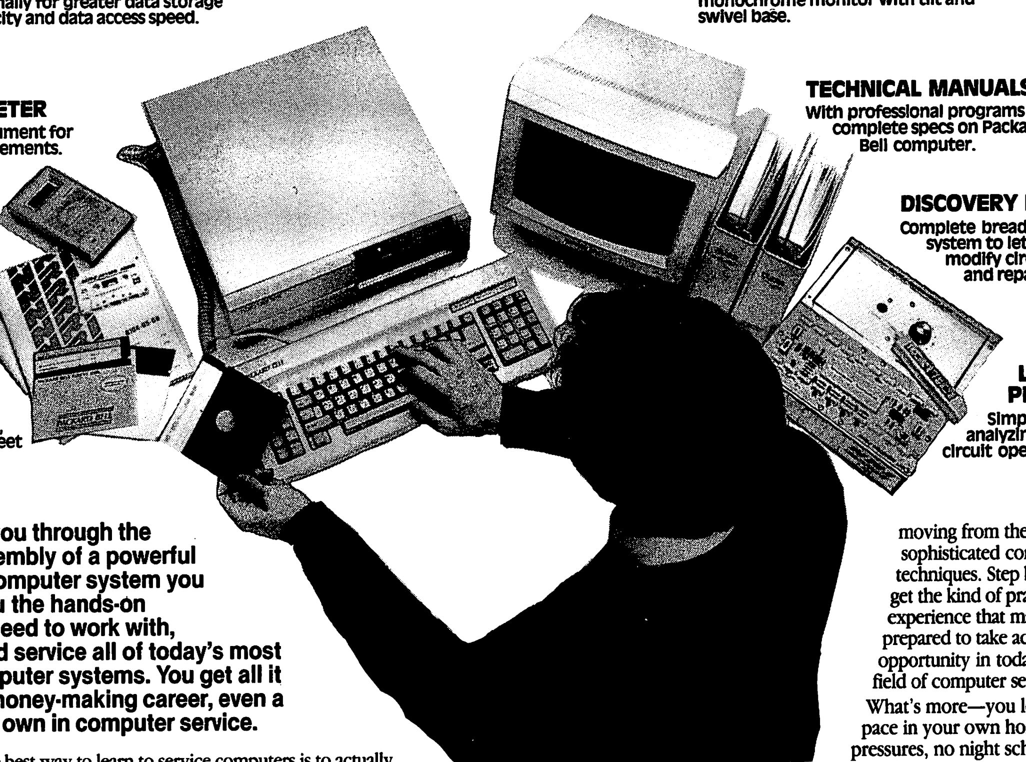
ALL THE ABOVE FEATURES, AND MUCH MORE!
PLUS A FREE UTILITY DISK w/SUPER EXPLODE! V4.1.
MAKE YOUR C-64, 64-C or C-128*, D-128* SUPER FAST and EASY to use.

SUPER TRACKER

Utilities Unlimited has done it again. At last an easy way to find out where the protection really is. **Super Tracker** will display the location of your drive head while you are loading a piece of software. This information will be very useful, to find where the protection is. **Super Tracker** has other useful options such as: track and half-track display, 8 and 9 switch, density display, write protect on/off. This incredible little tool is encased in a handsome box that sits on top of your drive. Works with all C/64/128 and most C/64 compatible drives. Some minor soldering will be required.

Introductory Priced at Just **$69.95**

No other training—in school, on the job, anywhere—shows you how to trouble-shoot and service computers like NRI



Only NRI walks you through the step-by-step assembly of a powerful XT-compatible computer system you keep—giving you the hands-on experience you need to work with, troubleshoot, and service all of today's most widely used computer systems. You get all it takes to start a money-making career, even a business of your own in computer service.

No doubt about it: The best way to learn to service computers is to actually *build* a state-of-the-art computer from the keyboard on up. As you put the machine together, performing key tests and demonstrations at each stage of assembly, you see for yourself how each part of it works, what can go wrong, and how you can fix it.

Only NRI—the leader in career-building, at-home electronics training for 75 years—gives you such practical, real-world computer servicing experience. Indeed, no other training—in school, on the job, *anywhere*— shows you how to trouble-shoot and service computers like NRI.

You get in-demand computer servicing skills as you train with your own XT-compatible system—now with 20 meg hard drive

With NRI's exclusive hands-on training, you actually build and keep the powerful new Packard Bell VX88 PC/XT compatible computer, complete with 512K RAM and 20 meg hard disk drive.

You start by assembling and testing the "intelligent" keyboard, move on to test the circuitry on the main logic board, install the power supply and 5¼" disk drive, then interface your high-resolution monitor. But that's not all.

Only NRI gives you a top-rated micro with complete training built into the assembly process

Your NRI hands-on training continues as you install the powerful 20 megabyte hard disk drive—today's most wanted computer peripheral—included in your course to dramatically increase your computer's storage capacity while giving you lightning-quick data access.

Having fully assembled your Packard Bell VX88, you take it through a complete series of diagnostic tests, mastering professional computer servicing techniques as you take command of the full power of the VX88's high-speed V40 microprocessor.

In no time at all, you have the confidence and the know-how to work with, troubleshoot, and service every computer on the market today. Indeed you have what it takes to step into a full-time, money-making career as an industry technician, even start a computer service business of your own.

No experience needed, NRI builds it in

You need no previous experience in computers or electronics to succeed with NRI. You start with the basics, following easy-to-read instructions and diagrams, quickly moving from the fundamentals to sophisticated computer servicing techniques. Step by easy step, you get the kind of practical hands-on experience that makes you uniquely prepared to take advantage of every opportunity in today's top-growth field of computer service.

What's more—you learn at your own pace in your own home. No classroom pressures, no night school, no need to quit your present job until you're ready to make your move. And all throughout your training, you have the full support of your personal NRI instructor and the NRI technical staff always ready to answer your questions and give you help whenever you need it.

Your FREE NRI catalog tells more

Send today for your free full-color catalog describing every aspect of NRI's innovative computer training, as well as hands-on training in robotics, video/audio servicing, electronic music technology, security electronics, data communications, and other growing high-tech career fields.

If the coupon is missing, write to NRI School of Electronics, McGraw-Hill Continuing Education Center, 4401 Connecticut Avenue, Washington, DC 20008.

PC/XT and XT are registered trademarks of International Business Machines Corporation

NRI School of Electronics
McGraw-Hill Continuing Education Center
4401 Connecticut Avenue, Washington, DC 20008

For career courses approved under GI bill
☐ check for details

☑ **Check one FREE catalog only**

- ☐ **Computers and Microprocessors**
- ☐ Robotics
- ☐ TV/Video/Audio Servicing
- ☐ Computer Programming
- ☐ Security Electronics
- ☐ Electronic Music Technology
- ☐ Basic Electronics
- ☐ Data Communications

Name (please print) Age

Address

City/State/Zip 0000-000

Accredited by the National Home Study Council

Transactor
The Magazine for Commodore Programmers

Volume 9, Issue 4

Departments and Columns

Reviews

About the cover: *ThunderBear* was created by Wayne Schmidt using *Kwikpaint* and *Artist 64*. Wayne describes the picture as follows:

"Inspired by Kwakiutl motifs (North West Coast Indians), the subject is a 'transformation' mask design worn by shamans during initiation and ceremonial events. During these events, the Thunderbird outer mask opens to reveal a great Bear spirit."

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. USPS Postmasters: send address changes to: Transactor, PO Box 338, Station C, Buffalo, NY, 14209.

Croftward 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., Subscriptions Department, 85 West Wilmot Street, Unit 10, 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.

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. Write to Transactor's Richmond Hill office to obtain a writer's guide.

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

Follow-ups and returns

Last issue this space was devoted to copy-protection. Specifically, I was upset that Commodore *GEOS* is protected and Apple *GEOS* isn't. Since that time I have discovered a very interesting thing: When first released, Apple *GEOS was* copy-protected but the response from the Apple community was so negative that Berkeley Softworks removed the protection! Since then, Apple *GEOS* sales figures have improved. Of course, some may contend that sales were stimulated primarily by the release of Apple *GeoPublish*, but my own feeling is that sales of both are augmented because of the absence of copy-protection.

This means there is hope! Write to Berkeley and tell them how you feel. Maybe we can get the situation changed. In the case of Apple *GEOS*, Berkeley Softworks has demonstrated that they are responsive to the user community. The ball is in *your* court now...

The response to Paul Bosacki's hardware project in 9:2 has been phenomenal. Seems like everybody wants more RAM! A sampling of the letters Paul has received (and Paul's replies) have been included in *Letters*. Also in this issue is an article by Richard Curcio suggesting a similar project for the C128. Included with that are the letters between Paul and Richard and some schematics from Richard. This is a topic that is not going to go away! Now if only the price of RAM would come down...

In 9:3 we were 'pushing the limits' with George Hug's *Toward 2400;* another article that was very well received. I don't know how many people have called about that. Some of them called to get the missing data statement line from the CIA test program (see *Bloops*). Some were interested in incorporating the routines in their own programs. At least the prices on 2400 bps modems have become affordable...

We're pushing the limits in 'fleshware' too. By the time you read this the *Transactor* editorial staff will be 33% larger. That's right, we have a new editor. Tim Grantham did eight *Amiga Dispatches* columns for us before Don Curtis started doing it when we started up *TransAmi*. Tim also recently started a new column in *TransAmi* called *Hard Copy*. Tim has been 'in deep' for quite a while with both the Amiga and the 8-bit machines. We're very pleased to welcome Tim back to *Transactor*. The extra manpower means we'll have time for other things, such as eating, sleeping and breathing.

Also returning this issue are *Letters* and *NewsBRK*. To those of you who wrote in about it or were just wondering, they haven't been discontinued. We just needed the space. Both departments are fairly weighty this time.

Malcolm D. O'Brien

LeTteRS

Please address letters to the editor to:
Letters, Transactor Magazine,
10-85 West Wilmot Street,
Richmond Hill, Ontario L4B 1K7.

A tale of woe... and a solution: I would like to thank you all for the complimentary copy of *Transactor* containing Joel M. Rubin's comparison review of a few Commodore macro assemblers; my *Buddy* included. Brian Hilchie once said to me, regarding a not so totally positive review of his C compiler, "Bad publicity is better than no publicity." Of course it wasn't a bad review, or unfair. But it still pissed me off and made me say some gosh darn bad words.

But before I launch into any more heart-rending whining or the reason why I felt compelled to say these bad words, I would like to share a glimpse of the software industry through the eyes of *Buddy*'s greatest fan and victim: its author; the idea being not only to egoize on a bit, but to try and piece together an admittedly cynical software publication theory to be considered by other users and hackers pure in spirit.

Every hacker has dreams (in addition to those of flying and being found inexplicably and embarrassingly naked in public places) of gaining recognition and reward for his awesome programming accomplishments. I (except for the flying dreams) am no different.

Before *Buddy*, I did a screen design and animation utility for games programmers called *64 Animator*. Although life on Earth (and perhaps my marriage) could well have carried on much the same if I had not, it wasn't bad, and every publisher that ever looked at it drew up a contract. Except Commodore. Commodore looked at it for about six months and then said they weren't in the software business anymore. I do believe however that some of the ideas in it may have found their way into the sprite routines on the 128. (Call me paranoid; call me a megalomaniac; just don't call me a lawyer.) Commodore (Canada) returned the sixty bucks I had put up to get them to look at it in the first place; thereby, in their expressed consideration, voiding our non-disclosure agreement.

Richvale Telecommunications (remember *Script 64*?) signed up for *Animator* next - then sat on it for about six months until they went under. Next it was Pro-Line's turn. Pro-line dinked around with it literally for years before releasing it to Spinnaker who released it back to them who released it back to me. Richard Evers' Northern Blue Marketing Inc. contracted to publish it next, and is currently doing so, and - get this - has actually sent me a royalty cheque. Richard and company and family, I love you; I really do.

Buddy assembler was Pro-Line's idea. The world owes *Buddy* to Pro-Line and OSAP (Ontario Student Assistance Program). I even made almost enough money the first year to, say, cover my hydro (I heat with oil). Still, it was better than the nothing I was accustomed to. Then Pro-Line decided to sign an agreement for all of their products with the big boys at Spinnaker down in Boston. And that was that. The first quarter I actually earned a negative royalty. Of course, they didn't make me pay; they let me owe it to them. Next quarter I paid Pro-Line back from royalties based on back collections from their own sales and I haven't seen a nickel since. And neither has Pro-Line from what I hear.

Anyone who bought Spinnaker's package (nobody according to my percentage) probably wondered why the name *Power Assembler* is displayed above C source code. After all, the assembler says "Buddy" when you run it; you have to type in "Bud" to invoke it; the manual refers to the "Buddy System" throughout; and C source would surely be a wonderful source - of syntax errors. But I would be willing to let them slide for seemingly sticking it in a used box; I would even be willing to ignore the fact that they have never advertised the product - if it were not for the following:

Spinnaker apparently does not believe in updating their releases. The version reviewed is as old as the proverbial hills. *Buddy* has been a true macro assembler for years; it has a vertical split-screen editor, superior memory management, speed, and display formatting, and even improved documentation (although some might still find it overly concise). Jim Butterfield, Liz Deal, Richard Evers, Miklos Garamszeghy, Darren Spruyt, John Lem and a zillion others gave me valuable feedback in bringing *Buddy* up to what is currently not available from Spinnaker today. Now, I know that hackers update their stuff hourly and a publisher can't always be restuffing boxes, but Spinnaker has had the latest and greatest for over a year. Just sitting there.

Now, I'll probably never get rich off *Buddy*, but it does give me a nice, warm, fuzzy feeling to know that someone out there is occasionally wowed by it and finds it useful. And it annoys the socks off me when giants sleep in the road.

I guess my advice to hackers in search of recognition and reward would be to seek out the smallest publisher you can find. Software publishers are like fish: the bigger they are, the harder they jerk you around. This is my theory; it is what I have come to believe.

Transactor has been my staunchest ally all along; always connecting me with the right people and the right information. Ever catering to my ever-starving ego. You guys saved my life. I cannot thank you enough.

I now spend my days coding medical systems in C on Kitchener Online Data's state of the art, 32-node, QNX network. I enjoy the work and the people. But the Commodore 64 was my first and is my deepest love. I feel bad for letting her languish sometimes, because if it were not for the Commodore 64 and *Transactor* I honestly believe that I would still be painting townhouses or washing pots for a living - instead of playing on the virtual circuits. All the best,

Chris Miller
2 Hilda Place, Kitchener, ON N2G 1K3

As you can see, this is not your average letter to the editor. Since it contains some provocative material, I phoned Chris to ask him if he would mind if we printed it. He said that he had written it figuring that we would publish at least some of it. Of course, the statements made and opinions expressed are Chris' own. ***Transactor*** *received a letter and published it, that's all.*

Since his main concern at this point, as expressed in the letter, is the unavailability of his upgrade, I made a suggestion and Chris agreed to it. I suggested that Chris supply the upgrade directly. Chris was concerned that users would think that he was just trying to scoop up some cash by writing his letter. I told him that I would say that the upgrade offer was my idea (and so it was). He also didn't want to charge so much that people would be put off and be denied the upgrade. We agreed that $10 was a reasonable figure and unlikely to dissuade anyone on the basis of cost.

Of course, the rights to ***Buddy*** *are held by Spinnaker and he can't be taking customers from them. So... if you send your original* ***Power Assembler*** *disk and $10 to the address printed above, Chris will send you the "latest and greatest" version of* ***Buddy*** *with some supplementary docs on disk.* ***I stress that it must be the original disk****. This way, Chris is not taking customers from Spinnaker. They make their money when you purchase* ***Power Assembler****. In effect, Chris is providing customer support for their product on "volunteer plus costs plus a little" basis. Since the product is not copy-protected, you won't have to do without while it's in the mail.*

Pushing the Limits: Paul Bosacki's article on expanding the C64's memory internally has been very well received, to put it mildly. Paul has received numerous letters and is still receiving them (and replying to them) as we go to press. In the space remaining we've included some of these letters along with Paul's responses. I've edited both the letters and responses somewhat to eliminate material which does not pertain directly to the article. - MO

Wanted: Non-volatile 1750: I read your article, "Care and Feeding of the C256" in *Transactor* 9:2 with great interest. I'm writing you in the hope that you may have an answer to a possible hardware fix for the 1750 REU. I use it as a CP/M RAMdisk on the C128. I'm told that it might be possible to convert the REU into a *non-volatile* RAMdisk. This apparently could be done by interrupting the power supply line from the main board inside the cartridge, inserting a "refresher circuit" (oscillator/regulator) and connecting this circuit to a direct and constant power supply.

Not a battery; used, for instance, by the *Quick Brown Box*. Since the REU uses dynamic and not static RAM, a battery wouldn't keep its charge very long, sustaining 512K of DRAM. But rather connecting the circuit to a transformer plug, of the kind used to power transistor radios when their batteries aren't being used. It was suggested that a plug supplying 7 volts would be about right.

Does it seem to you that this would be possible? If so, I'm not sure I'd attempt it myself, but with the proper instructions and a rough schematic, if needed, I'd happily hand it over to an experienced technician.

Carl Gabler, Van Nuys, CA

And I'm afraid that all I have is bad news for you. I too, have heard of such a hardware fix, but only in the "wouldn't it be great if" stages. When I first received your letter, I looked briefly into the problem, and though I believe it possible, I just haven't had the time to do anything about it. Your note outlined the problem exactly: external power supply, voltage regulator and a refresher of some sort.

*The 41256 DRAM has an onboard refresh counter, so all that's necessary in the refresher is a timer (the timing's tense) and a method of strobing *RAS while *CAS is held low. The power consumption would be high, but then we've an external power supply, so that's not a problem.*

Anyway, that's pretty much all I have to say on the topic right now. If I get any further with it, I'll let you know. On the other hand, if you manage to come up with anything, I'd appreciate hearing *about it. My REU could use the same treatment. Thank you for your interest. I'm sorry I couldn't be of more help.*

I just got *Transactor* 9:2 with your article in it. I am *very* interested in knowing how you managed to squeeze 1MB into a C64. Please send me any info you have on the procedure. Is the 1MB also GEOS compatible?

What other projects are in the works? How far could the 64 be expanded? Could another graphics chip be interfaced to improve resolution? Could two 64s be linked to common memory to do multitasking or parallel processing?

As you can see, I'm full of questions. Anything you could send to answer a few would be greatly appreciated. Thank you.

Dale Schoek, Hurst, TX

I stuffed a meg into the 64 using two very different techniques. The first, and perhaps least obvious - I expanded a 1764 REU to 512K. As you may know, this unit comes with 256K installed. Taking it to 512K is as simple as opening the unit and installing another bank of 41256's. The 150 nanosecond chips are the ones to use. They're designated with a -15 at the end of the chip number, i.e. 41256-15.

In order to install the extra RAM, you will have to clear the solder rings, but other than that, there's nothing else involved. Just a little proficiency with a soldering iron, and the usual anti-static precautions.

Now, concerning the 512K expansion board that I have installed in my 64: it now looks as though ***Transactor*** *will be publishing the 512K Upgrade article in an upcoming issue. Because of this, I'm back in development, trying to streamline the board. So until I test out the new board, I really would feel uncomfortable releasing that information. Keep your eyes open; it'll be in a* ***Transactor*** *soon.*

As well, the article will present the driver that will configure the extra RAM as a 1571 RAM DISK, as well as allow the 17xx REU's and my RAM expansion to function together. In the first article, it was one or the other. Not both.

What other projects are in the works? Mostly, I'm working on software right now, and the upcoming ***Transactor*** *article. I've also been looking at doing an internal RAM expansion to one meg, using one meg chips. It gets a little ridiculous though. Through banked RAM, there's no reason why you couldn't install megabytes of memory into a 64 (power supply considered). But when you consider that a 64 costs less than $175 and a full megabyte of RAM $320... well, you see what I mean. So, it stops somewhere, probably where the cost of the memory exceeds the orginal purchase price of the computer. 512K pushes it.*

Concerning some of your other questions here. There's no sense in adding another VIC chip to a 64. No, it wouldn't increase resolution. In fact, it's probably impossible due to the way the VIC handles the address and data buses and refresh timing and bus timing, etc. What I've always thought would be a really neat option is a new chip that's compatible with the earlier VIC but offers (through additional registers) higher resolutions, more colours, etc. That would be the way to go. But that's up to Commodore, so I doubt we'll see something like that. Although, I have heard rumours that Commodore is considering a new 8-bit computer that would be 64-compatible. Maybe a 'turbo' 64 with enhanced graphics and memory!

And lastly, concerning multitasking. The 512K board offers that option. The 256K mod, as you know, keeps a certain amount of memory common to each bank. The 512K mod allows you, through software, to turn off that option. What happens then is that, with a little bit of set-up, you have 8 absolutely separate 64K partitions. Each with its own stack, zpage, and Kernal vectors. The only thing that has to be handled is reloading the I/O chips on bank switch. You'll read all about it soon.

Some alternatives: I very much enjoyed reading your "C256" article in *Transactor* 9:2. To my knowledge you are the first to demonstrate a working conversion of this type for the C64. Now that you have done the hard part, the rest of us can just sit back and wait for some cheap RAM to come along.

If convenient, I would appreciate your sending me a list of any errors in the article. I assume that on page 71 the AEC switch setting under B(ii) should read "enabled when closed", but I would like to know about any other typos you may have found.

Congratulations on your successful project and fine articles. I look forward to future instalments.

Concerning typos in the article. You've caught the glaring one. The rest are, to my knowledge, just misspellings, and those are mine, not ***T****'s. The only other thing is an omission in the code. In the header declaration, the icon is missing. Apparently, the* ***T*** *is planning an errata. [The missing icon data is included as a "bloop" in this issue. - MO]*

b i t s

Got an interesting programming tip, a 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 subscription to Transactor.

Fast Graphics Primitives with SYMASS
Henry Cale, Roswell, New Mexico

Robert Huehn's excellent Fast Graphics Primitives program in the December 1989 issue (Volume 9 Issue 2) of *Transactor* works well after being assembled using SYMASS if the following minor changes are made to the assembly code: (1) Remove the comment after SYS 700 in line 100. (2) Enter **5700 .END** as the last line of the listing. (3) Change the code starting address to **290 *=$C000** or some other safe location. At $9000 the assembled code overwrites SYMASS's symbols table which propagates downward from SYMASS causing SYMBOL NOT FOUND errors.

Command Tails for the C64/C128
Noel Nyman, Seattle, Washington

At a recent user group meeting, a member asked me if it was possible to pass data to a program as part of the RUN statement. For example, CP/M allows you to type:

```
READ filename.ext
```

"READ" is the name of a program to be executed; CP/M loads and runs this program. It also makes available the data "filename.ext", called the 'command tail', to the program READ, which can do what it likes with the information. AmigaDOS, MS-DOS, and other operating systems also allow arguments to be passed to programs in this way.

At first, I said that you can't do that in Commodore BASIC, then I thought about it a bit and realized that you can, and quite easily!

When RETURN is pressed on a screen line, the entire line is transferred to an area of memory called the input buffer. The line is tokenized during the transfer: each BASIC keyword is replaced with a single-byte token.

```
RUN: "filename
```

On the C64, RUN can only be followed by a colon or a line number. The operating system ignores anything that follows the colon, but the bytes are still in the input buffer. All a program has to do is scan for the RUN token and the colon.

A zero byte indicates the end of the line in the buffer. Anything between the colon and the zero byte is data we can manipulate in any way we choose.

Below is a simple sequential file reading program that uses the 'command tail' to get the name of the file to read.

Anything between a colon following RUN and the end of the line is placed in BF$. Leading spaces and quote marks are discarded. If BF$ is null, there was no command tail and we ask the user for a file name.

You could check for numbers, MS-DOS style switches (slash), Unix-style options (hyphen), or anything you want. You can do it in machine language, or keep it in BASIC, which is probably fast enough.

The C128 makes the command even more useful. It allows RUN to also LOAD the program. So,

```
RUN "file reader": "filename
```

...will LOAD the program called "file reader" from drive 0 of device 8 and RUN it. File Reader then parses past the colon and puts "filename" in BF$. The main program then opens the requested file and displays its contents on the screen.

It's best to use quotes ahead of the file name to avoid tokenizing any BASIC commands that may be part of the name.

```
HG  100 rem parse the input buffer for a file
NC  110 rem name following run:
EO  120 rem by noel nyman
GP  130 :
PI  140 x=512 : rem start of input buffer
GH  150 xx=592: rem change to 673 for c128
FH  160 :  if peek(x)<138 and peek(x)>0 then x=x+1: goto 160: rem 138=run token
CJ  170 :  if peek(x)=0 goto 250: rem end of line in buffer
```

```
LN 180 :  if peek(x)=58 then x=x+1: goto 210: rem found a colon
MH 190 :  x=x+1: if x>xx goto 250: rem xx points to end of input buffer
DD 200 :  goto 180
KG 210 :  if peek(x)=32 and bf$="" then x=x+1: goto 210: rem skip leading spaces
GN 220 :  if peek(x)=34 then x=x+1: goto 220: rem skip quotes
PI 230 :  if peek(x)<>0 then bf$=bf$+chr$(peek(x)): x=x+1: goto 220
EG 240 :
IL 250 if bf$="" then input "enter file name: ";bf$: rem no arguments
IH 260 :
MM 270 open 15,8,15
OK 280 open 2,8,2,"0:"+bf$+",s,r"
FH 290 input#15,x,x$
PI 300 if x>19 then print "file name = "bf$: print "disk error"x,x$: goto 390
KK 310 :
LP 320 get#2,x$
PE 330 print x$;
IM 340 :
FP 350 wait 653,1,255: rem shift key pauses, shift/lock holds
MN 360 :
MN 370 if st=0 goto 320
AP 380 :
FO 390 close 2: close 15
EA 400 :
KJ 410 end
```

Quickie Flash Routine
R.J. Poulin, Frederick, Maryland

A short and sweet 'flash' routine for the C64 is always in demand. Here's one I use with my notice to a forgetful user to turn on his printer:

```
500 rem simple flash subroutine
510 rem r.j. poulin, frederick, md
520 f=0: print "press <return> when ready": print
530 print chr$(f)"turn on your printer!"chr$(145)
540 get a$: f=18-f: for i=1 to 250: next
550 if a$<>chr$(13) then 530
560 return
```

C128 ML Monitor Tricks
James Devlin, Decatur, Georgia

The following pokes will turn the RESTORE key into a monitor BRK key:

```
poke dec("0318"),peek(dec("0316"))
poke dec("0319"),peek(dec("0317"))
```

Enter the pokes and hit the RESTORE key (you don't need the STOP key). Your computer will BRK into the machine language monitor, just as if it had encountered a BRK instruction. And the best thing about this trick is that the program counter and registers displayed will reflect exactly what the computer was doing when you hit RESTORE!

This trick works by copying the monitor's BRK vector into the NMI vector, so that the non-maskable interrupt generated by tapping the RESTORE key forces a JMP to the monitor's BRK routine rather than the normal NMI routine.

Device Presence Checker
Paul Sawyer, Orangeville, Ontario

This machine language program will report on the presence of a given device number. You can use it to check if the user has his disk drive or printer plugged in and turned on before proceeding with an operation.

The program is relocatable; in this listing it resides at 49152. To use it, put the device number in location 252, then SYS 49152, and PEEK location 251; if the result is zero, the device is off or not connected.

If you check for a disk drive that is present, the drive error light will flash. Just initialize the drive and everything will be okay.

BASIC Loader

```
NG 100 rem  device presence checker
MJ 110 rem  by paul g sawyer
JA 120 rem  store device # in 252,
JO 130 rem  sys to routine (49152),
HK 140 rem  then peek(251). if value is
KJ 150 rem  0 then device is off,
LP 160 rem  1 then device is on.
IM 170 rem
NP 180 rem  if device checked is a disk
DK 190 rem  drive, initialize after calling
JM 200 rem  routine.
KP 220 for i=49152 to 49191:read x:poke i,x:next
BM 230 data 169,   0, 133, 251, 165, 252, 170, 160
HN 240 data   0,  32, 186, 255, 169,   1, 166, 251
BC 250 data 160,   0,  32, 189, 255,  32, 192, 255
OF 260 data 165, 144,  72, 165, 252,  32, 195, 255
BG 270 data 104,  48,   4, 169,   1, 133, 251,  96
```

Source code

```
NB 300 *=$c000
OJ 310 .opt oo
OM 320 device = $fc
NK 330 onflag = $fb
NH 340 lda #0: sta onflag
DH 350 lda device: tax: ldy #0
FJ 360 jsr $ffba   ;setlfs
CC 370 lda #1: ldx onflag: ldy #0
JI 380 jsr $ffbd   ;setnam
MG 390 jsr $ffc0   ;open
JK 400 lda $90: pha
FA 410 lda device: jsr $ffc3 ;close
CK 420 pla: bmi off
FI 430 on lda #1: sta onflag
DD 440 off rts
```

The ML Column

Crunching: from order to chaos

by Todd Heimarck

It doesn't seem to make sense. Crunching reduces the size of a file without reducing the amount of information there. How is that possible?

If you've used programs like ARC, you know that, however it works, it works. In this article, we'll look at the classic crunching algorithm called *Huffman encoding*.

Huffman encoding and other compression techniques usually reduce the size of a file. However, you can find mathematicians, computer scientists, and other theoreticians who can prove that compression algorithms can fail. Once in a while, you crunch a file and (surprise!) you get back something that's bigger than the original. That's the chance you take.

But that's a reasonable rule. If you owned a crunching algorithm that guaranteed a 20% reduction in size, you could run the program over and over again to compress a whole encyclopedia into one byte. You'd just run the output into the input until it shrank down to whatever size you wanted.

The theory: making chaos

The basic theory of crunching is that you look for patterns of order in the input file. You then replace the patterns with smaller codes that will later expand during the uncrunching process. The crunched file is more chaotic and more random (in an orderly way) than the source file.

As a very simple example, imagine that a text file contains 431 instances of the word "the". The word "qz" appears nowhere in the file, mainly because it's not an English word. If you search and replace "the" with "qz", you save 431 bytes. Two letters have replaced three letters. But you've made the file more random. At some point you run out of orderly patterns and you can't compress any more.

A benefit of crunching is that you save space, whether it's disk space or memory. A drawback is that it takes time to crunch and uncrunch.

Huffman encoding, which is capitalized because it's named after its inventor, typically reduces a file by 20 to 40 percent, sometimes more. It works best on text files, which abound with space characters between words. Text files also have a lot of characters in the a-z and A-Z range. They're predictable and orderly.

Enough theory. Let's crunch something.

How it works

The writer A.A. Milne, author of "Winnie the Pooh", wrote a poem that starts with these lines:

> Of all the Knights in Appledore
> The wisest was Sir Thomas Tom.
> He multiplied as far as four,
> And knew what nine was taken from
> To make eleven.

To introduce the concepts of Huffman coding, we will crunch those five lines. Including letters, punctuation, spaces, and five carriage returns, there are 143 bytes to crunch.

On the first pass of the Huffman program, we count each character's frequency. Next, we sort the letters from most common to least common. That list is used to create the Huffman codes, which are labeled HCODES in Table 1.

Columns 1-3 list the character, the PETASCII code, and the frequency. Columns 4 and 5 give the Huffman code and its length (note that the length varies in this example from 3 bits to 8 bits and that the most frequent characters have the shortest length). Columns 6 and 7 list the number of bits for the Huffman code and the number of bits for the eight-bits-in-a-byte ASCII code.

Take the entry for the letter 'A', for example. It looks like this:

```
A    193    2    111001    6    12    16
```

The PETASCII code is 193 and it appears two times in the sample text. The Huffman code for this particular example is

Table 1: Characters Sorted by Frequency

CHR	ASC	FREQ	HCODE	HLEN	HBITS	8BITS
spc*	32	23	000	3	69	184
e	69	14	110	3	42	112
a	65	10	0100	4	40	80
n	78	8	1000	4	32	64
s	83	8	1001	4	32	64
i	73	7	1011	4	28	56
l	76	6	1111	4	24	48
t	84	6	00100	5	30	48
o	79	6	00101	5	30	48
r	82	5	00111	5	25	40
h	72	5	01100	5	25	40
w	87	5	01101	5	25	40
m	77	5	01010	5	25	40
<*	13	5	01011	5	25	40
f	70	4	01111	5	20	32
T	212	4	10111	5	20	32
k	75	3	11101	5	15	24
d	68	3	001100	6	18	24
p	80	3	001101	6	18	24
.	46	2	011101	6	12	16
u	85	2	111000	6	12	16
A	193	2	111001	6	12	16
,	44	1	0111001	7	7	8
S	211	1	01110000	8	8	8
O	207	1	01110001	8	8	8
K	203	1	1010110	7	7	8
H	200	1	1010111	7	7	8
v	86	1	1010100	7	7	8
g	71	1	1010101	7	7	8

*** The space character and the carriage return are listed as spc and <.**

111001, which is 6 bits long (in another example, the code might be something completely different). In the crunched file, 2 times 6 is 12 (frequency times length). In a normal ASCII file, two 8-bit bytes would need 16 bits. For this character, we crunch 16 bits down to 12, a savings of 25%.

Comparing the HBITS column (630 bits) against the 8BITS column (1144 bits) shows that using Huffman codes should save 45%. Actually, as we'll see in a minute, the overhead needed for the code table wipes out the savings.

It's not at all obvious where the Huffman code 111001 for "A" came from. We'll get to that soon. But first, let's examine a coded line. Figure 1 shows the characters of the first line of the poem. The Huffman code from the table above is under each character. The bits are repackaged as 8-bit bytes, which would be written to the crunched file. The commas would not appear in memory or in the file; they're included to visually separate the bits.

Figure 1: Crunching "Of all the Knights of Appledore<"

```
O        f     spc  a    l     l    spc t    h      e      spc
-------- ----- ---- ---- ----- ---- --- ---- ------ ------ --- ---
01110001, 01111 000, 0100 1111, 1111 000 0,0100 0110,0 110 000
$71       $78        $4F        $F0        $46        $61

K        n     i    g        h      t     s     spc o      f
-------- ----- ---- -------- ------ ----- ----- --- ------ ------
1,010110 10,00 1011 10,10101 011,00 00100 1,001 000 00,101 01111,
  $5A      $2E        $AB      $09        $20        $AF

spc A       p      p       l     e   d       o       r    e   <
--- ------- ------ ------- ----- --- ------- ------- ---- --- ------
000 11100,1 001101 0,01101 111,1 110 0011,00 00101 0,0111 110 0,1011
$1C         $9A        $6F       $E3        $0A         $7C      $B?
```

Uncrunching

Now it's time to uncrunch the file, which you may notice is a lot more chaotic than it was before. The letter "l" repeats three times, for example, in the original line. Not one byte repeats in the crunched line.

Before we start to uncrunch, we should briefly review binary trees. A 'tree' is a data structure made up of nodes. There's always one node at the top. A node can do one of two things: it can terminate or it can branch. If it branches, it's called a *parent node* and the nodes below are called *children.*

In a binary tree, the parent nodes can have only two children - not three and not one. If you branch to the left, you find child 0. On the right is child 1. (Other kinds of trees can have more than two children, but we're using a strictly binary tree for Huffman codes.)

The Huffman codes listed above happen to fit into a binary tree that's illustrated in Figure 2.

To uncrunch the Huffman codes, we need to do something called "traversing the tree". The bit patterns of the first two bytes look like this:

```
01110001, 01111000
```

Start at the top of the tree and read the bits from left to right. First there's a 0, which means you move down to the left to child 0, which happens to be a parent node. Next a 1, so move down and right to child 1. Then right, right, left, left, left, and right. The eight node terminates in the letter "O". Print that letter (or send it to an output file) and go back to the top of the tree. One left and four rights takes us to the terminating node "f". Print it and go back to the top. The next traversal does three quick lefts and hits the space character.

You might ask the question "What if one code is 01 and another is 0101?" How do you know when to stop and when to continue? If you look at the tree, you'll see that a node

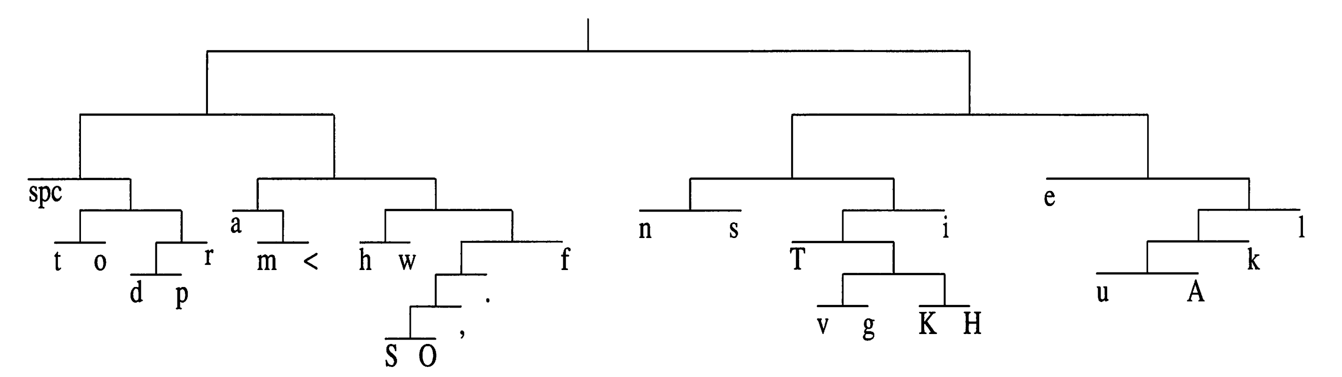


Figure 2: The binary tree

either terminates or has children. It's impossible to have both 01 and 0101. Since the space character starts with 000, no other characters start with 000x (where x is a 0 or 1).

How to grow a tree

Of course, the next question is how do you create a binary tree? Well, we want it to be 'weighted', which means we want the popular characters near the top (a short path) and the unpopular ones near the bottom (a longer path).

Let's start with a small example. Suppose you have a file with six characters distributed like this (the "t" means "terminating node"):

```
tE      50
tT      31
tA      15
tO      12
tN       8
tQ       8
```

After sorting the frequencies, you start building nodes from the bottom. Combine tN and tQ into a new node called p01 (the "p" means "parent node"), which has a frequency of 8+8 or 16. It replaces the second to the last node and we delete the last node:

```
tE      50
tT      31
tA      15
tO      12
p01     16
```

The tree is still sorted, except for the final node. Next we perform an insertion sort, to put that node in its right place:

```
tE      50
tT      31
p01     16 (children tN and tQ)
tA      15
tO      12
```

Combine the two bottom nodes tA and tO into p02, with a frequency of 27 and sort again:

```
tE      50
tT      31
p02     27 (children tA and tO)
p01     16 (children tN and tQ)
```

The process continues until only two nodes remain, which are then hooked into the node at the very top.

How to use the program

Say you want to crunch a sequential file called ZEBRA into a sequential file called Z.HUFF. Load the Huffman program into memory and run this program:

```
10 OPEN 2,8,2,"ZEBRA,S,R": SYS 49152: REM FIRST PASS
20 OPEN 3,8,3,"ZEBRA,S,R": OPEN 4,8,4,"Z.HUFF,S,W": SYS 49155
```

The file Z.HUFF is created. In addition, the original file size and the crunched size are printed on the screen.

To uncrunch, run this program:

```
10 OPEN 5,8,5,"Z.HUFF,S,R": OPEN 6,8,6,"OUTFILE,S,W": SYS 49158
```

It's important that the files be opened as numbers 2, 3, and 4 when you crunch and 5 and 6 when you uncrunch.

Program Notes

The program has three parts: EPASS1 (49152), EPASS2 (49155), and DECODE (49158).

The EPASS1 routine counts the bytes in COUNTEM, which stores the frequencies in the tables CFREQLO and CFREQHI (the C means "character"). Then it sorts the list in SORTEM, which in turn calls ISORT, the insertion sort routine. Then MAKETREE builds a tree. The node frequencies are stored in NFREQLO and NFREQHI. The codes are in NCODE. The parent nodes for characters are in CPNODE, for nodes they're in NPNODE.

EPASS2 won't work unless EPASS1 has been called first. It sends out a set of bytes that describe the tree. First, the total number of bytes is sent (two bytes), then the number of characters that branch left (the zeros) and each one's parent and name. Next, the children who branch right (ones), the nodes that are 0s, and the nodes that are 1s.

The overhead needed to describe the tree increases the size of the file. For example, the poem above has 143 characters, but only 29 are unique. The header needs 29 times 4 plus 2 bytes: 118 bytes in all. The 143 characters crunch down into 79 bytes, but when you add the overhead, the crunched file contains 197 bytes (compared to 143 in the original).

The Huffman codes follow the header. In the example above, the letter "n" has the code 1000. The tables are arranged by parent nodes, so to figure out the code, the program works backwards: 0, 0, 0, 1. When it gets to the top, it reverses the (reversed) bits to get 1000. The variable-length codes are packed into bytes of eight bits each and sent out to the crunched file.

The DECODE routine uncrunches Huffman codes. It sets up a table (CH0NAME, CH0TYPE, CH1NAME, and CH1TYPE). Note that this is a table of children, whereas the encoding routine used a table of parents. In the decoding process, the program gets bits one at a time until it gets a child that's a character. Then it goes back to the top of the tree.

How well does it work?

In some tests I ran, text files crunched best. A 15400-byte file crunched down to 9118 bytes (from 61 disk blocks to 36). That's a 41% savings. The PAL source code from this program, which is mostly text, reduced from 10126 bytes to 8218, a 19% savings. The *SpeedScript* program (all machine language) didn't crunch very well at all, probably because of the overhead. It shrank slightly, from 6153 bytes to 6042, a 2% savings.

"huff.src" - *source code in PAL format*

```
IJ  100 rem save"huff.src"
PD  110 sys700
IL  120 *=49152
KO  130 .opt oo
HM  140 fstat   = $90        ; file status
MH  150 getin   = $ffe4
IC  160 chkin   = $ffc6
OI  170 chrin   = $ffcf
AK  180 clrchn  = $ffcc
LA  190 close   = $ffc3
GN  200 chkout  = $ffc9
JP  210 chrout  = $ffd2
HC  220 bb      = $c500
LO  230 cfreqlo = bb+0       ; characters' frequency
AD  240 cfreqhi = bb+$100
JA  250 ccode   = bb+$200    ; code (00=0, ff=1)
JE  260 cpnode  = bb+$300    ; parent node
FO  270 nfreqlo = bb+$400    ; nodes' freq
OH  280 nfreqhi = bb+$500
MH  290 ncode   = bb+$600    ; code
NF  300 npnode  = bb+$700    ; parent (0=top)
LM  310 list    = bb+$800    ; sorted list
PG  320 type    = bb+$900    ; type 00=char, ff=node
AK  330 hbits   = bb+$a00    ; my own stack
JP  340 ch0type = bb+0       ; reused variable space--child 0 type
EM  350 ch0name = bb+$100    ; child 0 name
AB  360 ch1type = bb+$200
NN  370 ch1name = bb+$300
OK  380 jmp epass1           ; encode pass one
CL  390 jmp epass2           ; encode pass two
KL  400 jmp decode           ; decode
AB  410 ;
CC  420 epass1 = *
JP  430         ldx #2
KP  440         jsr chkin    ; open channel 2 for input
JO  450         jsr zeromem  ; zero out memory
BM  460         jsr countem  ; count the bytes
FM  470         lda #2
FF  480         jsr close
NP  490         jsr clrchn   ; close up channel 2
LI  500         jsr sortem   ; sort the list
AI  510         jsr maketree ; build the tree
GK  520         jsr node0    ; the tip of the tree
OP  530         rts
CJ  540 ;
HO  550 zeromem = *
LB  560         lda #0
JA  570         tay
OH  580 zeloop  sta cfreqlo,y
GG  590         sta cfreqhi,y
MG  600         sta ccode,y
HM  610         sta cpnode,y
LI  620         sta ncode,y
EA  630         sta type,y
EC  640         sta list,y
FE  650         dey
JA  660         bne zeloop
HN  670         sta filelen
IM  680         sta filelen+1
DA  690         sta numchar
PB  700         sta numnode
KJ  710         inc numnode  ;  save node 0 for the top
ML  720         rts
AF  730 ;
OK  740 countem = *
HG  750            jsr chrin        ; get a character from disk
HI  760            tax              ; index by .x
GM  770            inc cfreqlo,x    ; one more in that slot
OP  780            bne bytecount
PK  790            inc cfreqhi,x    ; if 0, then inc the high byte
JL  800 bytecount  inc filelen
IJ  810            bne cotest
GD  820            inc filelen+1
NE  830 cotest     ldy fstat        ; file status (0 = more to come)
JM  840            beq countem      ; go back for more bytes
JK  850            ldx filelen
OD  860            lda filelen+1
PP  870            jsr $bdcd
NP  880            lda #62
BG  890            jsr chrout       ; print length, >
LK  900            rts              ;
EA  910 ;
```

```
DE  920 sortem = *
NO  930         ldy #0
IL  940         sty listlen    ; used by the isort routine
OI  950         sty lc
GH  960 soloop  ldy lc
KL  970         lda cfreqlo,y
GI  980         ora cfreqhi,y  ; check if freq <> 0
ML  990         beq nochar     ; if eq, then no characters
IA  1000        tya            ; add it to the list
FG  1010        ldy numchar
DD  1020        sta list,y     ; this is the ascii code
EG  1030        inc numchar    ; one more character
HN  1040        jsr isort      ; insertion sort
AP  1050        inc listlen    ; the list has one more member
BG  1060 nochar inc lc
NH  1070        bne soloop     ; keep going with lc 0 to 255
EC  1080        rts
JE  1090 isort = *
KG  1100        ldy listlen    ; length of the list
GF  1110        bne is01       ;
BJ  1120        rts            ; if = 0, skip this
ME  1130 is01   lda list,y
KN  1140        sta islist
JE  1150        tax
IN  1160        lda type,y
GJ  1170        sta istype     ; save these values
EE  1180        bne anode      ; if <>0, it's a node
DJ  1190        lda cfreqlo,x
FG  1200        sta islo
BJ  1210        lda cfreqhi,x
FM  1220        sta ishi       ; save frequencies
EM  1230        jmp is02       ; go compare them
JF  1240 anode  lda nfreqlo,x
HJ  1250        sta islo
OM  1260        lda nfreqhi,x
HP  1270        sta ishi       ; save frequencies
FO  1280 is02 = *
FH  1290        dey            ; count backward in the list
GN  1300        ldx list,y
OG  1310        lda type,y
NP  1320        bne anode2     ; another node
PB  1330        lda cfreqlo,x
DP  1340        sta testlo
NB  1350        lda cfreqhi,x
HP  1360        sta testhi
AG  1370        jmp is03
OA  1380 anode2 lda nfreqlo,x
FC  1390        sta testlo
KF  1400        lda nfreqhi,x
JC  1410        sta testhi
CH  1420 is03 = *
JO  1430        lda ishi
EM  1440        cmp testhi     ; compare
KG  1450        bcc insert     ; insert in the list here
LO  1460        bne is04       ; keep looping, maybe
FD  1470        lda islo
HD  1480        cmp testlo     ; not sure, so check low byte
GP  1490        beq insert     ; if equal, insert
IP  1500        bcc insert     ; if islo < testlo, insert
KK  1510                       ; else drop through
MM  1520 is04   cpy #0
LE  1530        bne is02       ; if .y = 0, drop through to insert
PL  1540        dey
PC  1550 insert = *
IP  1560        iny
CM  1570        sty tempy      ; save the value
HC  1580        cpy listlen
FM  1590        bne doit
MC  1600        rts
```

```
DI  1610 doit   ldy listlen    ; start at the end
DP  1620 isloop dey
EM  1630        lda list,y
IE  1640        iny
GB  1650        sta list,y
HD  1660        dey
GN  1670        lda type,y
AH  1680        iny
IC  1690        sta type,y
PF  1700        dey
GO  1710        cpy tempy
ID  1720        bne isloop
KO  1730        lda islist
AH  1740        sta list,y
HG  1750        lda istype
OG  1760        sta type,y
GN  1770        rts
KG  1780 ;
LC  1790 maketree = *
HH  1800         ldx numchar
JM  1810         dex
PD  1820         stx listlen
HH  1830 mamain  ldy listlen
HI  1840         jsr fixcn     ; fix the codes & nodes for y and y-1
GB  1850         jsr fixfreq   ; fix the new node's frequency
CA  1860         jsr addnode   ; add the node to the list
EK  1870         jsr isort     ; sort it
NJ  1880         inc numnode
LO  1890         lda listlen
OJ  1900         cmp #1
KI  1910         bne mamain    ; quit when only two nodes remain
MG  1920         rts
GP  1930 fixcn   = *
NH  1940         ldy listlen
LJ  1950         lda #$ff      ; this means code = 1
CL  1960         jsr fixsr     ; set the code/node
NG  1970         dey
HK  1980         lda #0
FF  1990         jsr fixsr     ; code = 0 on the left
ML  2000         rts
PH  2010 fixsr   ldx type,y
ML  2020         beq tsachar   ; itsa char
AL  2030         ldx list,y
BN  2040         sta ncode,x   ; it's a node
HC  2050         lda numnode
IH  2060         sta npnode,x
CA  2070         rts
JM  2080 tsachar ldx list,y
LD  2090         sta ccode,x
JF  2100         lda numnode
PJ  2110         sta cpnode,x
ED  2120         rts
JD  2130 fixfreq = *
FE  2140          ldy listlen
CB  2150          ldx type,y
IE  2160          beq anotchar ; another char
MD  2170          ldx list,y
MH  2180          lda nfreqlo,x
HO  2190          sta low1
KH  2200          lda nfreqhi,x
FL  2210          sta hi1
PE  2220          jmp ahead
KP  2230 anotchar ldx list,y
NK  2240          lda cfreqlo,x
DC  2250          sta low1
LK  2260          lda cfreqhi,x
BP  2270          sta hi1
IF  2280 ahead    dey
OJ  2290          ldx type,y
IL  2300          beq itschar  ; another char
IM  2310          ldx list,y
IA  2320          lda nfreqlo,x
HH  2330          sta low2
GA  2340          lda nfreqhi,x
EE  2350          sta hi2
JO  2360          jmp addem
```

```
AC 2370 itschar  ldx list,y
JD 2380          lda cfreqlo,x
DL 2390          sta low2
HD 2400          lda cfreqhi,x
AI 2410          sta hi2
IJ 2420 addem    ldx numnode
EP 2430          clc
DK 2440          lda low1
DK 2450          adc low2
CN 2460          sta nfreqlo,x
LH 2470          lda hi1
KH 2480          adc hi2
KN 2490          sta nfreqhi,x
AL 2500          rts
EE 2510 ;
MA 2520 addnode = *
GG 2530       dec listlen
BN 2540       ldx listlen
DL 2550       lda #$ff
EJ 2560       sta type,x     ; type of a parent is always a node
PC 2570       lda numnode
FE 2580       sta list,x     ; add the node number to the list
KA 2590       rts
OJ 2600 ;
GD 2610 node0 = *
JI 2620       ldy #1
IA 2630       ldx list,y
NA 2640       lda #$ff
GH 2650       sta ncode,x
PE 2660       lda #0
KG 2670       sta npnode,x   ; the parent is node 0 at the top
DD 2680       dey
EE 2690       ldx list,y
IK 2700       sta ncode,x
CA 2710       sta npnode,x
MI 2720       rts
AC 2730 ;
FD 2740 epass2 = *
NA 2750       ldx #4
AJ 2760       jsr chkout     ; channel 4 for writing
NL 2770       lda #0
DI 2780       sta outlen
GN 2790       sta outlen+1   ; zero out file length
DP 2800       jsr header     ; send the header bytes
HO 2810       jsr encfile    ; send the encoded file
HP 2820       lda #4
DI 2830       jsr close
JA 2840       lda #3
HJ 2850       jsr close
DD 2860       jsr clrchn
LP 2870       ldx outlen
FL 2880       lda outlen+1
DG 2890       jsr $bdcd      ; print crunched length
AE 2900       rts
EN 2910 ;
DA 2920 header = *
NG 2930       lda filelen
DF 2940       jsr chrout
IG 2950       lda filelen+1
DE 2960       jsr chrout     ; length of input file
FO 2970       ldy #0
LO 2980       ldx #0
HL 2990 char0 lda cfreqlo,x
AD 3000       ora cfreqhi,x  ; is this char in file
LL 3010       beq head0      ; if no freq, doesn't exist
PJ 3020       lda ccode,x
JH 3030       bne head0      ; ignore $ff
EJ 3040       txa
ND 3050       sta hbits,y
CF 3060       iny            ; push on temp stack
```

```
AH 3070 head0 inx
IJ 3080       bne char0
JM 3090       tya
IJ 3100       pha
NP 3110       jsr chrout
NE 3120       jsr sendchar    ; send # of 0children and then names
AL 3130 ;
PI 3140       ldy #0
FJ 3150       ldx #0
AO 3160 char1 lda ccode,x
JN 3170       beq head1       ; ignore $00
AC 3180       txa
JM 3190       sta hbits,y
ON 3200       iny             ; push on temp stack
OP 3210 head1 inx
FC 3220       bne char1
FF 3230       tya
EC 3240       pha
JI 3250       jsr chrout
KN 3260       jsr sendchar    ; send # of 1children and then names
MD 3270 ;
LB 3280        ldy #0
DC 3290        ldx #1
CK 3300 pnode0 lda ncode,x
DJ 3310        bne head2       ; ignore $ff
MK 3320        txa
FF 3330        sta hbits,y
KG 3340        iny             ; push on temp stack
MI 3350 head2  inx
DL 3360        cpx numnode
KD 3370        bne pnode0
LO 3380        tya
KL 3390        pha
PB 3400        jsr chrout
BL 3410        jsr sendnode    ; send # of 0nodes and then names
CN 3420 ;
BL 3430        ldy #0
JL 3440        ldx #1
LD 3450 pnode1 lda ncode,x
NP 3460        beq head3       ; ignore $00
CE 3470        txa
LO 3480        sta hbits,y
AA 3490        iny             ; push on temp stack
EC 3500 head3  inx
JE 3510        cpx numnode
CN 3520        bne pnode1
BI 3530        tya
AF 3540        pha
FL 3550        jsr chrout
IE 3560        jsr sendnode    ; send # of 1nodes and then names
IG 3570 ;
PE 3580         ldy #4
GC 3590 addloop pla
GI 3600         clc
FH 3610         adc outlen
LM 3620         sta outlen
JB 3630         lda #0
PJ 3640         adc outlen+1
FP 3650         sta outlen+1
HA 3660         dey
KH 3670         bne addloop
MA 3680         asl outlen
ID 3690         rol outlen+1
KO 3700         clc
HO 3710         lda outlen
BH 3720         adc #6
JD 3730         sta outlen
HI 3740         lda #0
NA 3750         adc outlen+1
DG 3760         sta outlen+1
GK 3770         rts
KD 3780 ;
```

```
HM 3790 sendchar dey
JE 3800          ldx hbits,y
FA 3810          lda cpnode,x
PI 3820          jsr chrout   ; send parent's name
KK 3830          txa
MC 3840          jsr chrout   ; send name
HG 3850          cpy #0
HL 3860          bne sendchar
KA 3870          rts
OJ 3880 ;
ND 3890 sendnode dey
NK 3900          ldx hbits,y
EH 3910          lda npnode,x
DP 3920          jsr chrout   ; send parent's name
OA 3930          txa
AJ 3940          jsr chrout   ; send name
LM 3950          cpy #0
BE 3960          bne sendnode
OG 3970          rts
CA 3980 ;
AP 3990 encfile = *
CK 4000          inc filelen+1
FK 4010          lda #8
LO 4020          sta outbits  ; 8 bits in a byte
OH 4030 enloop   jsr walkup   ; walk up the tree
NO 4040          jsr walkdown ; output the byte
IM 4050          dec filelen
OE 4060          bne enloop
DM 4070          dec filelen+1
OD 4080          bne enloop   ; keep going while the characters are coming
FK 4090          ldx outbits
NG 4100          cpx #8
OG 4110          beq finish
GH 4120 lastone  asl outbyte
JN 4130          dex
NN 4140          bne lastone
FI 4150          ldx #4
CA 4160          jsr chkout
PH 4170          lda outbyte
OG 4180          jsr chrout   ; send the last one
HO 4190          inc outlen
DL 4200          bne finish
HA 4210          inc outlen+1
EC 4220 finish   rts
MP 4230 ;
KI 4240 walkup = *
HO 4250          ldx #3
PJ 4260          jsr chkin
OL 4270          jsr chrin
MM 4280          tax          ; get the next input byte
NA 4290          ldy #0
PJ 4300          lda ccode,x
JC 4310          sta hbits,y
IH 4320          iny          ; first code
NA 4330          lda cpnode,x
FE 4340          beq upout    ; if parent is zero, exit
GO 4350 uploop   tax
GO 4360          lda ncode,x
FG 4370          sta hbits,y
ED 4380          iny          ; get the code and save it
EF 4390          lda npnode,x
PG 4400          bne uploop   ; branch if not parent 0
GE 4410 upout    dey
AD 4420          rts
EM 4430 ;
LM 4440 walkdown = *
HH 4450          lda hbits,y
NC 4460          rol
BM 4470          rol outbyte  ; build a byte a bit at a time
KM 4480          dec outbits
KG 4490          beq gotabyte ; if outbits = 0, 8 bits are ready
```

```
FI 4500 downtest dey
CH 4510          cpy #255
CD 4520          bne walkdown
GO 4530          rts          ; end of walkdown
CD 4540 ;
JO 4550 gotabyte sty tempy
PB 4560          ldx #4
MJ 4570          jsr chkout
JB 4580          lda outbyte
FD 4590          jsr chrout   ; send a character
OB 4600          ldy tempy
EG 4610          inc outlen   ; increment length of output file
LD 4620          bne reset8
LK 4630          inc outlen+1
DL 4640 reset8   lda #8
HL 4650          sta outbits
JN 4660          bne downtest ; branch always
EL 4670 ;
FM 4680 decode = *
DK 4690          ldx #5
DP 4700          jsr chkin    ; channel 5 is input
GH 4710          jsr chrin
JK 4720          sta filelen
KI 4730          jsr chrin
EK 4740          sta filelen+1
AJ 4750          inc filelen+1
OP 4760 chi0     jsr chrin    ; how many child0's
LK 4770          beq chi1
KA 4780          sta lc       ; loop counter
GP 4790 delp1    jsr chrin
LI 4800          tax
KN 4810          jsr chrin
HM 4820          sta ch0name,x
JM 4830          lda #0
BB 4840          sta ch0type,x
HC 4850          dec lc
BK 4860          bne delp1
MH 4870 ;
KH 4880 chi1     jsr chrin    ; how many child1's
JD 4890          beq par0
CI 4900          sta lc       ; loop counter
AH 4910 delp2    jsr chrin
DA 4920          tax
CF 4930          jsr chrin
CE 4940          sta ch1name,x
BE 4950          lda #0
MI 4960          sta ch1type,x
PJ 4970          dec lc
KB 4980          bne delp2
EP 4990 ;
HF 5000 par0     jsr chrin    ; how many parent0's
FL 5010          beq par1
KP 5020          sta lc       ; loop counter
KO 5030 delp3    jsr chrin
LH 5040          tax
KM 5050          jsr chrin
HL 5060          sta ch0name,x
LI 5070          lda #$ff
BA 5080          sta ch0type,x
HB 5090          dec lc
DJ 5100          bne delp3
MG 5110 ;
EN 5120 par1     jsr chrin    ; how many parent1's
KK 5130          beq bitter
CH 5140          sta lc       ; loop counter
EG 5150 delp4    jsr chrin
DP 5160          tax
CE 5170          jsr chrin
CD 5180          sta ch1name,x
DA 5190          lda #$ff
MH 5200          sta ch1type,x
PI 5210          dec lc
MA 5220          bne delp4
EO 5230 ;
```

```
PG  5240 bitter = *
NG  5250          lda #0
OP  5260          sta node
HD  5270 outloop  ldx #5
LJ  5280          jsr chkin
KL  5290          jsr chrin
EK  5300          sta huffer      ; got 8 bits
JL  5310          lda #8
PD  5320          sta numbits
GJ  5330 inloop   ldy node        ; the node is the parent
IK  5340          rol huffer      ; get a bit into carry
OP  5350          bcs itsa1       ; if cs, the bit is 1
BD  5360 itsa0 = *
HH  5370          lda ch0name,y ; get the name of child 0
MJ  5380          sta node      ; who is the new parent/node
PC  5390          ldx ch0type,y ; does it terminate
LE  5400          beq printit   ; yes, go print
MM  5410 nextbit = *
AG  5420          dec numbits
MK  5430          bne inloop
JN  5440          beq outloop
AM  5450 ;
HJ  5460 itsa1 = *
AO  5470          lda ch1name,y ; get the name of child 1
AA  5480          sta node      ; who is the new parent/node
GJ  5490          ldx ch1type,y ; does it terminate
PK  5500          beq printit   ; yes, go print
JD  5510          bne nextbit
GA  5520 ;
BA  5530 printit = *
HP  5540          ldx #6
AH  5550          jsr chkout
MO  5560          lda node
JJ  5570          jsr chrout
CM  5580          dec filelen
EN  5590          bne tothetop
NL  5600          dec filelen+1
KO  5610 tothetop lda #0
BD  5620          sta node        ; back up to node 0 at the top
JP  5630          lda filelen
LA  5640          ora filelen+1
FM  5650          bne nextbit
CJ  5660 ;
KM  5670 cleanup = *
FC  5680          lda #5
PK  5690          jsr close
LD  5700          lda #6
DM  5710          jsr close
PF  5720          jsr clrchn
OE  5730          rts
CO  5740 ;
MC  5750 filelen  = *      ; length of src file
EL  5760 numchar  = *+2    ; # of chars
OJ  5770 numnode  = *+3    ; # of nodes
IE  5780 lc       = *+4    ; loop counter
NA  5790 listlen  = *+5    ; length of list, used by isort routine
MA  5800 islist   = *+6    ; temporary storage for list, type, freqlo,
JC  5810 istype   = *+7    ; and freqhi
OA  5820 islo     = *+8
DA  5830 ishi     = *+9
NK  5840 testlo   = *+10   ; islo/hi is tested against testlo/hi
GG  5850 testhi   = *+11   ; for insertion sort
EO  5860 tempy    = *+12   ; temp storage for .y
FH  5870 low1     = islo
NM  5880 hi1      = ishi
EK  5890 low2     = testlo
DH  5900 hi2      = testhi
OL  5910 outlen   = *+13   ; output file length
OI  5920 outbyte  = *+15   ; the (crunched) huffman byte to send out
JA  5930 outbits  = *+16   ; number of bits (when it=8, the byte gets sent)
ME  5940 huffer   = islo   ; huffman byte (when uncrunching)
EF  5950 node     = ishi   ; the child node (either another node or a char)
OF  5960 numbits  = testlo ; number of bits
```

```
LB  100 rem  generator for "object"
FB  110 n$="object": rem  name of program
HH  120 nd=1178: sa=49152: ch=173041
```

(for lines 130-260, see the standard generator on page 5)

```
HG  1000 data  76,   9, 192,  76,  32, 194,  76, 172
NP  1010 data 195, 162,   2,  32, 198, 255,  32,  38
MB  1020 data 192,  32,  81, 192, 169,   2,  32, 195
EI  1030 data 255,  32, 204, 255,  32, 120, 192,  32
BB  1040 data  62, 193,  32,   6, 194,  96, 169,   0
CF  1050 data 168, 153,   0, 197, 153,   0, 198, 153
PA  1060 data   0, 199, 153,   0, 200, 153,   0, 203
EB  1070 data 153,   0, 206, 153,   0, 205, 136, 208
HI  1080 data 232, 141, 154, 196, 141, 155, 196, 141
OL  1090 data 156, 196, 141, 157, 196, 238, 157, 196
MH  1100 data  96,  32, 207, 255, 170, 254,   0, 197
EJ  1110 data 208,   3, 254,   0, 198, 238, 154, 196
MF  1120 data 208,   3, 238, 155, 196, 164, 144, 240
DK  1130 data 232, 174, 154, 196, 173, 155, 196,  32
DA  1140 data 205, 189, 169,  62,  32, 210, 255,  96
MI  1150 data 160,   0, 140, 159, 196, 140, 158, 196
LJ  1160 data 172, 158, 196, 185,   0, 197,  25,   0
HP  1170 data 198, 240,  16, 152, 172, 156, 196, 153
BP  1180 data   0, 205, 238, 156, 196,  32, 161, 192
BA  1190 data 238, 159, 196, 238, 158, 196, 208, 224
CI  1200 data  96, 172, 159, 196, 208,   1,  96, 185
OI  1210 data   0, 205, 141, 160, 196, 170, 185,   0
GI  1220 data 206, 141, 161, 196, 208,  15, 189,   0
LC  1230 data 197, 141, 162, 196, 189,   0, 198, 141
PN  1240 data 163, 196,  76, 209, 192, 189,   0, 201
MP  1250 data 141, 162, 196, 189,   0, 202, 141, 163
IA  1260 data 196, 136, 190,   0, 205, 185,   0, 206
PI  1270 data 208,  15, 189,   0, 197, 141, 164, 196
PI  1280 data 189,   0, 198, 141, 165, 196,  76, 245
AD  1290 data 192, 189,   0, 201, 141, 164, 196, 189
CA  1300 data   0, 202, 141, 165, 196, 173, 163, 196
IG  1310 data 205, 165, 196, 144,  17, 208,  10, 173
PH  1320 data 162, 196, 205, 164, 196, 240,   7, 144
BE  1330 data   5, 192,   0, 208, 196, 136, 200, 140
NO  1340 data 166, 196, 204, 159, 196, 208,   1,  96
CH  1350 data 172, 159, 196, 136, 185,   0, 205, 200
ED  1360 data 153,   0, 205, 136, 185,   0, 206, 200
LF  1370 data 153,   0, 206, 136, 204, 166, 196, 208
FJ  1380 data 234, 173, 160, 196, 153,   0, 205, 173
IP  1390 data 161, 196, 153,   0, 206,  96, 174, 156
LO  1400 data 196, 202, 142, 159, 196, 172, 159, 196
CJ  1410 data  32,  95, 193,  32, 141, 193,  32, 244
MA  1420 data 193,  32, 161, 192, 238, 157, 196, 173
IA  1430 data 159, 196, 201,   1, 208, 231,  96, 172
MN  1440 data 159, 196, 169, 255,  32, 110, 193, 136
IM  1450 data 169,   0,  32, 110, 193,  96, 190,   0
FI  1460 data 206, 240,  13, 190,   0, 205, 157,   0
DF  1470 data 203, 173, 157, 196, 157,   0, 204,  96
GA  1480 data 190,   0, 205, 157,   0, 199, 173, 157
JE  1490 data 196, 157,   0, 200,  96, 172, 159, 196
JJ  1500 data 190,   0, 206, 240,  18, 190,   0, 205
FO  1510 data 189,   0, 201, 141, 162, 196, 189,   0
GD  1520 data 202, 141, 163, 196,  76, 182, 193, 190
JD  1530 data   0, 205, 189,   0, 197, 141, 162, 196
MB  1540 data 189,   0, 198, 141, 163, 196, 136, 190
GO  1550 data   0, 206, 240,  18, 190,   0, 205, 189
JP  1560 data   0, 201, 141, 164, 196, 189,   0, 202
CA  1570 data 141, 165, 196,  76, 221, 193, 190,   0
HF  1580 data 205, 189,   0, 197, 141, 164, 196, 189
EF  1590 data   0, 198, 141, 165, 196, 174, 157, 196
AN  1600 data  24, 173, 162, 196, 109, 164, 196, 157
FE  1610 data   0, 201, 173, 163, 196, 109, 165, 196
IL  1620 data 157,   0, 202,  96, 206, 159, 196, 174
GK  1630 data 159, 196, 169, 255, 157,   0, 206, 173
LE  1640 data 157, 196, 157,   0, 205,  96, 160,   1
MH  1650 data 190,   0, 205, 169, 255, 157,   0, 203
EJ  1660 data 169,   0, 157,   0, 204, 136, 190,   0
NF  1670 data 205, 157,   0, 203, 157,   0, 204,  96
JF  1680 data 162,   4,  32, 201, 255, 169,   0, 141
BB  1690 data 167, 196, 141, 168, 196,  32,  74, 194
HN  1700 data  32,  27, 195, 169,   4,  32, 195, 255
```

```
BM 1710 data 169,   3,  32, 195, 255,  32, 204, 255
HA 1720 data 174, 167, 196, 173, 168, 196,  32, 205
DE 1730 data 189,  96, 173, 154, 196,  32, 210, 255
II 1740 data 173, 155, 196,  32, 210, 255, 160,   0
CJ 1750 data 162,   0, 189,   0, 197,  29,   0, 198
LA 1760 data 240,  10, 189,   0, 199, 208,   5, 138
LL 1770 data 153,   0, 207, 200, 232, 208, 235, 152
GE 1780 data  72,  32, 210, 255,  32, 245, 194, 160
AA 1790 data   0, 162,   0, 189,   0, 199, 240,   5
DO 1800 data 138, 153,   0, 207, 200, 232, 208, 243
HH 1810 data 152,  72,  32, 210, 255,  32, 245, 194
KB 1820 data 160,   0, 162,   1, 189,   0, 203, 208
NB 1830 data   5, 138, 153,   0, 207, 200, 232, 236
PF 1840 data 157, 196, 208, 240, 152,  72,  32, 210
HM 1850 data 255,  32,   8, 195, 160,   0, 162,   1
BE 1860 data 189,   0, 203, 240,   5, 138, 153,   0
KG 1870 data 207, 200, 232, 236, 157, 196, 208, 240
LF 1880 data 152,  72,  32, 210, 255,  32,   8, 195
PL 1890 data 160,   4, 104,  24, 109, 167, 196, 141
PK 1900 data 167, 196, 169,   0, 109, 168, 196, 141
AM 1910 data 168, 196, 136, 208, 237,  14, 167, 196
GP 1920 data  46, 168, 196,  24, 173, 167, 196, 105
MK 1930 data   6, 141, 167, 196, 169,   0, 109, 168
CJ 1940 data 196, 141, 168, 196,  96, 136, 190,   0
JO 1950 data 207, 189,   0, 200,  32, 210, 255, 138
LP 1960 data  32, 210, 255, 192,   0, 208, 238,  96
CJ 1970 data 136, 190,   0, 207, 189,   0, 204,  32
OF 1980 data 210, 255, 138,  32, 210, 255, 192,   0
CP 1990 data 208, 238,  96, 238, 155, 196, 169,   8
OG 2000 data 141, 170, 196,  32,  84, 195,  32, 122
JB 2010 data 195, 206, 154, 196, 208, 245, 206, 155
ME 2020 data 196, 208, 240, 174, 170, 196, 224,   8
BA 2030 data 240,  25,  14, 169, 196, 202, 208, 250
BG 2040 data 162,   4,  32, 201, 255, 173, 169, 196
HA 2050 data  32, 210, 255, 238, 167, 196, 208,   3
KJ 2060 data 238, 168, 196,  96, 162,   3,  32, 198
EN 2070 data 255,  32, 207, 255, 170, 160,   0, 189
HC 2080 data   0, 199, 153,   0, 207, 200, 189,   0
JL 2090 data 200, 240,  13, 170, 189,   0, 203, 153
IB 2100 data   0, 207, 200, 189,   0, 204, 208, 243
KJ 2110 data 136,  96, 185,   0, 207,  42,  46, 169
FJ 2120 data 196, 206, 170, 196, 240,   6, 136, 192
GK 2130 data 255, 208, 239,  96, 140, 166, 196, 162
CF 2140 data   4,  32, 201, 255, 173, 169, 196,  32
FN 2150 data 210, 255, 172, 166, 196, 238, 167, 196
LJ 2160 data 208,   3, 238, 168, 196, 169,   8, 141
IB 2170 data 170, 196, 208, 218, 162,   5,  32, 198
FA 2180 data 255,  32, 207, 255, 141, 154, 196,  32
IO 2190 data 207, 255, 141, 155, 196, 238, 155, 196
HN 2200 data  32, 207, 255, 240,  23, 141, 158, 196
FO 2210 data  32, 207, 255, 170,  32, 207, 255, 157
PM 2220 data   0, 198, 169,   0, 157,   0, 197, 206
KN 2230 data 158, 196, 208, 236,  32, 207, 255, 240
NA 2240 data  23, 141, 158, 196,  32, 207, 255, 170
FN 2250 data  32, 207, 255, 157,   0, 200, 169,   0
OP 2260 data 157,   0, 199, 206, 158, 196, 208, 236
NB 2270 data  32, 207, 255, 240,  23, 141, 158, 196
LC 2280 data  32, 207, 255, 170,  32, 207, 255, 157
AC 2290 data   0, 198, 169, 255, 157,   0, 197, 206
AC 2300 data 158, 196, 208, 236,  32, 207, 255, 240
DF 2310 data  23, 141, 158, 196,  32, 207, 255, 170
CM 2320 data  32, 207, 255, 157,   0, 200, 169, 255
EE 2330 data 157,   0, 199, 206, 158, 196, 208, 236
OO 2340 data 169,   0, 141, 163, 196, 162,   5,  32
JI 2350 data 198, 255,  32, 207, 255, 141, 162, 196
MF 2360 data 169,   8, 141, 164, 196, 172, 163, 196
NN 2370 data  46, 162, 196, 176,  18, 185,   0, 198
MN 2380 data 141, 163, 196, 190,   0, 197, 240,  20
KI 2390 data 206, 164, 196, 208, 232, 240, 214, 185
KE 2400 data   0, 200, 141, 163, 196, 190,   0, 199
OC 2410 data 240,   2, 208, 236, 162,   6,  32, 201
FK 2420 data 255, 173, 163, 196,  32, 210, 255, 206
GL 2430 data 154, 196, 208,   3, 206, 155, 196, 169
AO 2440 data   0, 141, 163, 196, 173, 154, 196,  13
GC 2450 data 155, 196, 208, 204, 169,   5,  32, 195
IL 2460 data 255, 169,   6,  32, 195, 255,  32, 204
GC 2470 data 255,  96
```

FOR COMMODORE 64/128

RAPIDOS

AND FOR CP/M COMPUTERS

So all your friends have IBM's and Macs?

Now you can feel sorry for them.

THE SUPERFAST PARALLEL DOS FOR THE COMMODORE 64/128 AND 1541

You're a Commodore owner. You've got great colors and fantastic sound, dynamic features that IBM and Mac owners can only dream of. You've got it all - except speed.

Well, you don't have to be left in the dust anymore! Lawrence Hiler, one of the original "Basement Boys", brings you the most exciting development in the history of Commodore computers - RapiDos! Packed with features, RapiDos will transform your slug into a bullet - just take a look at the chart below!

Get your RapiDos, then invite your IBM and Mac friends over to see your "new" super computer. But be nice - make sure they're sitting down before you blow them away!

Sample RapiDos Features:

- ALL disk access commands execute at superfast speeds!
- Designed to support multiple drives!
- On board DOS Wedge / Screen Dump / M-L Monitor!
- Centronics parallel printer support available!
- Fully Commodore compatible & your cartridge port is left free!

RapiDos Professional Features:

- Gives even faster disk access than RapiDos!
- 8K RAM track buffering and hardware GCR conversion!
- Provides 40 track extension (749 blocks FREE)!
- Adds 20 NEW disk commands (lock files, rename disk, etc)!

SPECIAL: RAMBOard™ capabilities are built in! RapiDos Pro can use Maverick™ parameters to backup some of the newest, hottest titles on the market today!

Parallel Utilities Disk Available For Both Versions - Only $19.95

Function	Normal DOS	RapiDos	RapiDos Pro	Your System
Load 202 blocks	128 seconds	15 seconds	3 seconds	?
Save 202 blocks	196 seconds	98 seconds	8 seconds	?
Format 35 tracks	90 seconds	24 seconds	18 seconds	?

RapiDos - Only $49.95 / RapiDos Professional - Only $119.95

These prices are for a standard C-64 and 1541 drive. RapiDos is availabe for other Commodore configurations, but prices do vary - contact us for details. RapiDos requires a socketed kernal ROM U4 in your C-64. RapiDos is easily upgradeable to the Professional version. The RapiDos Professional drive controller is ©1987 mts data GbR, the creators of the best parallel systems in Europe!

• A Hiler Design •

THE MSD MASS DUPLICATOR

If You're Looking For Speed, You've Come To The Right Place

No one will deny that a stock Coorvette is a fast car. But in the hands of a knowledgeable expert, it can get faster. Much faster. Lawrence Hiler is a Commodore hardware expert. The first time he saw an MSD SD-2, he liked what he saw. But he knew it could go faster.

Much faster.

The MSD Mass Duplicator is firmware that turns an ordinary SD-2 into a professional, high speed disk copying system of unequaled performance. What kind of performance? How about:

- 15 second backup of standard formatted disks!
- 18 second full GCR QuickNibble for archiving protected disks!
- 9 second disk format - even for two disks at the same time!

Not bad, eh? But that's not all - our AutoCopy ROM Option allows you to use your SD-2 WITHOUT having it connected to your Commodore computer at all! Your MSD is transormed into a DEDICATED High speed duplication machine - a machine so fast and reliable that it is the same system we use here at Software Support to produce our own commercial software!

If You're lucky enough to own an MSD SD-2, you're just a simple, solderless, plug-in step away from owning the finest, fastest high speed disk duplicator we've ever seen. Why waste any more time - order Mass Dupicator today!

THE MSD MASS DUPLICATOR/ONLY $39.95

Special Bonus-Now Includes Our Dynamic AutoCopy ROM Option FREE!

TURBO 64

Life In The Fast Lane

What does a Turbo 64 cartridge have in common with a Lamborgini? They're both fast. Expensive. And definitely not for everyone. For most people, the Turbo 64 cartridge would be overkill. But for the few who can handle it, it can be as satisfying as a fast ride in a hot car.

Just plug the Turbo 64 cartridge in, and the on board 65816 processor takes over. This is the same high speed 8/16 bit chip that's found in the Apple IIGS. You can adjust the clock speed from 100khz (1/10 of normal) to a blistering 4.5mhz (4 1/2 times normal speed)! Your games, graphics, and applications (including GEOS) will flow. Get Turbo 64 - and get into the passing lane!

Turbo 64/$199.00

C-64 Burst ROM / C-128 Burst ROM

Exceed The Speed Limit

If you own a C-64 with a socketed Kernal ROM U-4, you're in luck. With our C-64 Burst ROM chip installed, you can access the burst mode of a Commodore 1571 or 1581 disk drive. This means that you can load 100 blocks in just 6 seconds on a 1571 (even double sided format), and 4 seconds on a 1581 drive! In addition, you get super fast access of directories and sequential and relative files. There's even a DOS wedge built in!

The C-128 version provides the same features as found on the C-64 version, and allows the C-128 to run at burst speed when in the 64 mode. Get the version for your machine, and start speeding today!

C-64 Burst ROM/Only $39.95 C-128 Burst ROM/Only $49.95

Please Note: The Kernal ROM U4 Chip In Your Computer Must Be Socketed To Accept A Burst ROM Chip

SOFTWARE SUPPORT INTERNATIONAL

PLEASE READ BEFORE ORDERING: We accept money orders, certified checks, VISA, M/C and, Discover. Previous Software Support customers may use C.O.D. and personal checks. Orders shipped to U.S.A. (48 states), F.P.O., A.P.O., or possessions, please add $3.50 per order for S & H. U.S. shipping is by UPS ground in most cases. FAST 2nd DAY AIR available: add $1.00 per pound additional (U.S. 48 states only). Alaska or Hawaii (all orders shipped 2nd day air), please add $7.50 per order for S & H. C.O.D. available to U.S. customers only (50 states): add $2.75 along with your S & H charges per order. Canadian customers may calculate the S & H charges by including $4.00 (minimum charge) for the first two pieces of SOFTWARE and $1.00 for each additional piece per shipment. All monies must be submitted in U.S. funds. Canadians must call or write for hardware shipping charges. Foreign customers must call or write for shipping charges. Defective items are replaced at no charge if sent postpaid. All in stock orders are processed within 24 hours. U.S. SOFTWARE orders over $100 will be shipped 2nd Day Air at our regular $3.50 S & H charge (48 states only). Washington residents please add 7.6% additional for Sales Tax. All prices subject to change. All sales are final unless authorized by management.

Mail your order to: Software Support, Int.
2700 NE Andresen Road Vancouver, WA 98661

Or call our toll-free order line at 1-800-356-1179. 9am-5pm Pacific time Monday-Friday
After hours orders accepted at (206) 695-9648 7 days a week.
Technical support available. Call (206) 695-9648. 9am-5pm Pacific time. Monday-Friday.
Orders Outside USA call (206) 695-1393.

VISA MasterCard DISCOVER

DEALERS — WE HAVE THE SUPPORT YOU'RE LOOKING FOR!

The Edge Connection

Experiences with a RAM expansion unit

by Joel M. Rubin

On Boxing Day (not a holiday here), having looked far and wide for the mythical 1750 512K RAM Expansion Unit for the Commodore 128 (which appears to be on eternal back-order), I grabbed a 1700 for $90 (US). Contrary to the promises on the box, I received neither a "new" CP/M disk nor the RAMDOS software; but, since the price was right and I already had both pieces of software, I did not scream too loudly.

The newest version of the U.S. C128 Kernal ($FF80 contains the identifying value $01 and the DMA Kernal jump at $FF50 jumps to the patch area at $CF80) fixes three bugs with DMA access. First of all, as previously noted in this magazine, it temporarily shuts off maskable interrupts (SEI) which could force the C128 into BANK 15 before the deferred DMA access is consummated. (If bit 4 of $DF01 is not set - which is the way BASIC does it - a DMA process is not carried out until the next write to $FF00. But, the NMI and IRQ routines write to $FF00, setting BANK 15 so, if they happen at the wrong time, the DMA transfer will take place from BANK 15, even if you wanted BANK 0.)

Of course, one cannot shut off non-maskable interrupts, so be careful with the RESTORE key and pseudo-RS-232 activities.

Secondly, if you request a DMA activity using BANK 1 RAM, the new Kernal routine temporarily switches the VIC/DMA RAM BANK register (bit 6 of $D506) to BANK 1. Therefore, with new Kernal 128s, contrary to the REU manual, you do not have to do this yourself when you do a BASIC FETCH, STASH or SWAP involving BANK 1 RAM - just use the BASIC "BANK 1" statement.

However, the new Kernal call does not temporarily switch to SLOW mode, so you will still have to switch to 1 MHz yourself. (Actually, DMA operations seem to work correctly in FAST mode, but the 8502 gets confused at the end of the process and tends to drop into the monitor.)

Finally, under the old routine, there was some trouble with the I/O chips ($D000-$DFFF) always being in context during a Kernal DMA operation, causing errors (and likely, crashes) when you tried to use DMA operations on RAM under the I/O page. The new Kernal routine fixes that.

```
old dma-call ($ff50 - $f7a5):

.ff7a5  bd f0 f7 lda $f7f0,x ; get configuration value
.ff7a8  29 fe    and #$fe    ; mask out i/o bit
.ff7aa  aa       tax         ; move configuration value to x-register
.ff7ab  4c f0 03 jmp $03f0   ; jump to common ram dma

new dma-call ($ff50 - $cf80):

.fcf80  ad 06 d5 lda $d506   ; save the vic ram bank on the stack
.fcf83  48       pha
.fcf84  5d f0 f7 eor $f7f0,x
.fcf87  29 3f    and #$3f
.fcf89  5d f0 f7 eor $f7f0,x
.fcf8c  8d 06 d5 sta $d506   ; turn it to bank 1 for bank 1 operations
.fcf8f  bd f0 f7 lda $f7f0,x
.fcf92  aa       tax
.fcf93  08       php         ; save the status on the stack
.fcf94  78       sei         ; shut off irq's (but not nmi's, alas)
.fcf95  20 f0 03 jsr $03f0   ; do dma
.fcf98  28       plp         ; cli if irq's previously turned on
.fcf99  68       pla
.fcf9a  8d 06 d5 sta $d506   ; recover old vic ram bank
.fcf9d  60       rts

.003f0  ae 00 ff ldx $ff00
.003f3  8c 01 df sty $df01
.003f6  8d 00 ff sta $ff00
.003f9  8e 00 ff stx $ff00
.003fc  60       rts
```

Word processors which allow you to load the whole dictionary into the REU and then automatically check your spelling as you type, such as *Paperclip III/128*, and the newest version of *WordPro 128*, won't do this autochecking if you only have 128K of REU, but *Paperclip* can load all the overlays you want into the REU. *GEOS 2.0* and *GEOS 128* won't allow you to use a 128K REU as a RAMdisk, but you *can* configure a 128K REU for fast rebooting after exiting to BASIC, and/or to speed up GEOS applications by allowing the VLIR overlays to be stored in the REU. Fast rebooting does not require the copy-protected GEOS key disk - only a disk with the *deskTop*. **Geoprogrammer**'s superdebugger, as sophisticated a debugger as is found on many mainframes, works fine on a 128K REU.

Of course, all but the earliest versions of CP/M allow the use of a 128K REU as a RAMdisk, device M:. Not all CP/M software supports device M:, however, you *can* get around this. First (due to Ken Flippo writing in November 1988

FOGHORN, FOG, P.O. Box 3474, Daly City, California 94015 U.S.A., Telephone (415) 755-2000) you can poke the address of the RAM disk handler into the DPH pointer address of the logical disk drive you want the RAM disk to be. For example, in the 8 December 1985 BIOS, disk drive C: has its table entry at 0FBD5h and the RAM disk handler is at 0FB96h. Like the 6502, the Z80 likes its words low-byte first. Ergo,

```
conf poke fbd5=96fb
```

would set/change device C: from device 10, drive 0, to the RAM disk. 0FBD1h is device A:, 0FBD3h is device B:, and so on. If you want, you can put this line in a *profile.sub* file, so it automatically happens on boot. Or, if you are a fanatic, you can either find the patch address in your version of *CPM+.SYS* and modify it or, if you have the source code for your BIOS, (the source code for one of the December 1985 BIOSs is included in the CP/M development package) you can change it and reassemble it. When I examined *CPM+.SYS* with **sid** and the C128 native-mode monitor (which, unlike **sid**, has a hunt function), I did not find an immediately obvious patch spot, so this chore is left as an exercise to the reader.

According to Miklos Garamszeghy's CP/M+ memory map (*Transactor* 9:2 or *Twin Cities 128* issue 21) the entry for device A: is at 0FBD1h in all versions of CP/M+ which support the RAM expander - all December and May versions. (August versions use a different address but don't support the REU.) However, if you have a version other than 8 December, you may wish to check the value of the REU handler. Use:

```
conf dump fbe9
```

and substitute the first two values for 96FB above.

One C128 CP/M user, J. Waltrip, made several modifications to his BIOS. The source and object codes are available, among other places, on FOG disk 186. Mr. Waltrip has the RAMdisk as device D:, and he also eliminated virtual drive E: and 40-column support.

An REU also provides a way of transferring files between Commodore DOS and CP/M. To copy a file from CP/M to Commodore DOS, **pip** it to the REU and reboot to Commodore DOS. On a 1700, the file begins at bank 0, $0800. The directory entry, near the beginning of the REU, shows how long the file is, in the standard CP/M directory entry method. With the 1700, each allocation unit is 1K, and the same should be true with the 1764. However, since each allocation unit has a one-byte entry in CP/M, if you have a 1750, each allocation unit must be 2K (512K/256 entries).

Going the other way, first boot CP/M. To create a directory entry up to 64K, use **save** to save a file to the REU. If you have a bigger file, you can create more than one file, each in multiples of the REU allocation unit (1K or 2K). Now, reboot to Commodore DOS, put your file in 17xx memory, beginning at $0800, bank 0, and reboot CP/M. Since the RAMdisk survives a reboot, your file will now be in device M:. If you have more than one file, you can pip them together - making sure, if they are not text files, that **pip** is in binary mode.

(For relatively short files, there is an easier method not involving the REU - from Commodore DOS to CP/M, load the file in BANK 1, beginning at $1C00, and then boot CP/M+ and use the CP/M+ program **save** twice. To get a file into Commodore DOS, use **sid** or **ddt** on it. Move the area from $0100 to $03FF to a higher area in memory (since $0100-$03FF of BANK 1 are ordinarily invisible in 128 mode). Now, hit stop-reset and end up in the monitor. Your file is in BANK 1.)

I can't get my 1700 to work with my old **Cardco** multi-cartridge board. I don't know if this is just a lack of physical contact or that the I/O expansion lines are buffered in some inappropriate way. I have trouble using the **Cardco** board with some (but not all) bank-switched cartridges which use the $DE00 or $DF00 I/O expansion spaces. **Simon's BASIC** worked fine with the **Cardco** board and a 64, but not with the **Cardco** board and a 128. (It works fine on the 128 without the **Cardco** board.) On the other hand, both the 128 and 64 modes of **Cinemaware**'s *Warpspeed* (my favourite cartridge, right now) work with the **Cardco** board, and the 64 mode uses the $DE00 I/O space for bank switching. Unfortunately, *Warpspeed* also uses the $DF00 I/O space, which is the same address as the RAM expansion controller chip, so there is no hope of using both *Warpspeed* and a 17xx at the same time - even with a different multi-cartridge board.

[Stop the presses! This just in from Joel:] The problem with my 1700 REU and **Cardco** cartridge turned out to be entirely physical - the 1700 works fine with it *if* you remove the plastic outer shell of the 1700. However, because of the $DF00 conflict, you still can't use the 1700 with *Warpspeed.*

Interrupts and the C128 80-column chip

There has been some question as to just when you can use an interrupt routine which writes to the C128 VDC chip. I've been playing with this a bit, and my conclusion seems to be that as long as the main program is not in an actual routine which reads or writes to the VDC chip, you are safe - *provided that you save and restore registers $12 and $13*, the RAM address registers.

For well-behaved programs (programs which stay with the normal text screens and only use the screen editor ROM routines to read or write to the VDC) you should be safe as long as the high byte of the program counter is *not* $CD. If you are using a program which does weird things to the VDC chip using its own RAM-based routines, such as *GEOS 128* or *BASIC 8*, all bets are off.

```
tsx
lda $107,x
cmp #$cd
bne ok
jmp $fa65 ; old irq
...
OK,  your code here
```

If you want to be a bit safer, you can make sure that the high byte of the PC is not between $C400 and $CFFF (The last page and a half of $C000 is a patch area so there might be something having to do with the VDC in the future). Since the computer spends much of its time in a waiting loop around $C200, this is much better than $C000-$CFFF. I have included an interrupt-driven clock routine which will run in either forty or eighty column text mode.

Hey, wanna buy a cheap orphan?

I notice that at least one mail order house (American Design Components, P.O. Box 220, Fairview, NJ 07022, telephone (201) 941-5000 or (800) 524-0809) is getting rid of Plus/4s for $49.95. These are returned machines, but they have been tested. What can you do with a Plus/4 (no anatomical suggestions, please)?

There is, or was, a Plus/4 users' group at P.O. Box 1001, Monterey, California. My local Toys-R-Us, just down the road from Wyatt Earp's tombstone, carries a word processor and an accounting package for the Plus/4. There are ads for various programs, including an assembler, in the pages of *Commodore Computing International*. The assembler is sold by a British software house, **Supersoft**, whose products are sold in the U.S. by **Skyles Electric Works**, so one might check with them. The Plus/4 can run many BASIC 7.0 graphics programs written for the 128.

While the Plus/4 only has 40 columns, it does have a 6551 UART, so it might be used for transferring files to/from Commodore DOS to other machines at a high baud rate. (*Paperclip III/128*'s telecommunications module and *GeoLaser* both have 9600 baud settings. So, if you do some CIA bit twiddling, rather than using the Kernal routines, it should be possible to use the C128, and maybe even the C64, at 9600 baud. I suspect that even with the fanciest bit twiddling, you'd still have to shut off the C128/C64 40-column screen to achieve such speeds, and you might need a goodly number of nulls if you want to scroll the C128 80-column screen.)

Moving forward with the 8-bit line

Finally, a reply to *Info* magazine's rumour column. According to the January/February issue, Commodore was thinking of putting out a super-64 or super-128, and one of the questions they were asking was, "How compatible must such a machine be?" *Info* said, "100%". But this would preclude such goodies as a 65816, a built-in UART at $DE00, a built-in 1750, a built-in 3.5" disk drive, et alia. I think that if Commodore's 8-bit line is to survive, it must, like the Apple //GS, move forward with the times. If it does, it will continue to get third party support. If it doesn't, it will, like the Atari 8-bit line, become just a cartridge game machine.

By the way, if Commodore is interested in competing with cartridge game machines, such as the Nintendo, it could bring out the Max Machine. (This was a game machine with which the C64 is upward compatible. It was never brought out in North America.) This could easily be sold for $50 (US) list and would have all the sound and graphics facilities of the C64. And, it would give us Commodore 8-bit hackers a new almost pirate-proof installed base for which to program games.

Clock.src: *C128 interrupt-driven clock routine for 40 or 80 column text mode.*

```
********************************
*  clock program for c'128     *
*  works in 40 or 80-column    *
*        text mode.            *
*  (c) 1989 joel m. rubin      *
********************************

col40_80 = $d7      ; current screen is 40-columns if 0,
                    ; 80-columns if 128

writdat = $cdca     ; .a => write .a to vdc data register (31)
writreg = writdat+2 ; .a, .x => write .a to vdc register #.x

rddat = $cdd8       ; read vdc data (31) register in .a
rdreg = rddat+2     ; .x => read vdc register #.x in .a

oldirq = $fa65      ; standard c'128 irq routine
irqv   = $0314      ; irq vector

time = $dc08        ; cia clock #1

 org $1300

 sei
 lda #<newirq
 sta irqv
 lda #>newirq
 sta irqv+1
 cli
 rts

*============================
* put 2 bcd digits (.a) on    *
* current screen at current   *
*       position              *
*============================

printit tay
        lsr
        lsr
        lsr
        lsr
        ora #"0"    ; double quotes here indicates most
                    ; significant bit set, so reversed
        jsr writit
        tya
print2  and #$0f
        ora #"0"
        jmp writit

*============================
* put a screen value at current*
*  position on current screen. *
*                              *
* datum in .a                  *
* for 40-column screen,        *
*  screen position in .x       *
*============================

writit pha
       lda col40_80
       bpl :40

       pla
       jmp writdat
```

```
:40    pla
       sta $400,x
       inx
       rts

newirq lda col40_80
       bpl do40

*==============================
* for 80 columns only, make   *
* sure that we're not doing   *
* anything with the vdc right *
* now--$c400 to $cfff no good. *
*                             *
* could probably get away with *
* $cd00 to $cdff in bank 15   *
* but there is a reference to *
* $d600 in $c500 page.        *
*                             *
* also, there is a patch area *
* at the end of $ce00 page    *
* to $cfff which could        *
* contain vdc references in   *
* the future.                 *
*==============================

col80 tsx
      lda $107,x
      cmp #$c4
      blt ok
      cmp #$d0
      bge ok

      jmp oldirq

*==============================
* for 80 columns only,        *
* put the values of vdc       *
* register $12 and $13 on     *
* the stack.                  *
*                             *
* since we are going to write *
* to the beginning of the     *
* 80 column screen, we are    *
* going to set these registers *
* to 0.                       *
*==============================

ok    ldx #$12
      jsr rdreg
      pha
      lda #0
      tay
      jsr writereg
      inx
      jsr rdreg
      pha
      tya
      jsr writereg

*==============================
* common routines for         *
* 40 columns and 80 columns   *
*==============================

do40  ldx #0
      lda time+3    ; hours + am/pm
      pha
      and #$1f      ; hours
      jsr printit
      lda #":"
      jsr writit
      lda time+2    ; minutes
      jsr printit
      lda #":"
      jsr writit
      lda time+1    ; seconds
      jsr printit
      lda #"."
      jsr writit
      lda time      ; tenths of seconds
      jsr print2
      lda #" "
      jsr writit
      pla           ; get back am/pm flag
      bmi pm
      lda #"a"&"?"
      hex 2c        ; skip next two bytes
pm    lda #"p"&"?"
      jsr writit
      lda #"m"&"?"
      jsr writit

      lda col40_80
      bpl fin40

*=============================
* 80 columns only--fill in   *
* vdc attribute ram with the *
* current color.             *
*=============================

      lda #8
      ldx #$12
      jsr writereg
      inx
      lda #0
      jsr writereg
      lda $f1
      and #$f
      ldy #13
:1    jsr writdat
      dey
      bne :1

*=============================
* 80 columns only--take old  *
* values of vdc registers    *
* $13 and $12 off the stack. *
*=============================

      ldx #$13
      pla
      jsr writereg
      dex
      pla
      jsr writereg
endit jmp oldirq

*=============================
* 40 columns only--fill in   *
* color ram with current     *
* cursor color.  if we are   *
* in lower/upper mode, turn  *
* the lower case reversed    *
* "am" or "pm" to "am" or "pm".*
*=============================

fin40 ldx #12
      lda $f1
:1    sta $d800,x
      dex
      bpl :1
      lda $a2c
      cmp #22       ; lower/upper mode
      bne endit
      lda $40b
      ora #$40
      sta $40b
      lda #"m"
      sta $40c
      bne endit
```

The C64 Power C Shell

With notes on modifying the shell for REU users

by Adrian Pepper

In a previous article ("On The C Side", *Transactor*, Volume 9, Issue 1), I hinted at the ability to use Power C with a RAM disk in a 1764 REU, but gave no details. Using the RAM disk software provided with the 1764 requires a 256 byte page of regular C64 RAM to be set aside for the interface to the RAM disk software. In the standard Power C environment, it is not possible to do this.

What follows will give some background to the explanation of making the RAM disk software work with Power C. It digresses a bit, but this should help lead to better understanding of the Power C environment in general. Even readers who don't use Power C may find this interesting and helpful, especially if they are considering purchasing it.

Overview

The Commodore 64 Power C shell is a machine language program that loads at $801, and occupies the memory almost to $1800. It is designed to work easily with one or two drives at a time. It recognizes a *work* drive, where user files (source, object and executable) should be, and a *system* drive, where files such as the different passes of the compiler, and object libraries are expected to be. These may be two drives in the same (dual drive) device, or drives in separate devices. The default configuration when the shell is run is suitable for a single drive system, and has both assigned to the same drive (device 8, drive 0). This configuration is usable, but necessitates a lot of disk swapping when compiling and linking programs.

Once running, the shell reads lines of user input from the screen. It parses the lines into words (series of non-blank characters), gathering them as the 'arguments' to the command to be executed. Generally, each word becomes a separate argument; the following characters cause certain exceptions:

" will turn a series of characters, including blanks, into a single argument, until a closing double-quote is found.

< will use the word following it as the name of a sequential file. It will look for the file on the work disk, and open it as *standard input* for the command executed in response to the command line. Neither the '<' nor the file name are actually passed as arguments to the command.

> will use the word following it as the name of a sequential file. It will attempt to create the file on the work disk, to receive *standard output* for the command executed. A special case is '>>', which will cause standard output to be directed to the printer.

The first argument found is taken as the name of a command. First, the shell checks to see if this is one of its own built-in commands:

- **bye**: exit to BASIC.
- **l**: list directory of work disk.
- **ls**: list directory of system disk.
- **rm**: remove (scratch) a file.
- **mv**: move (rename) a file.
- **pr**: copy a file on the work disk to standard output.
- **disk**: send arbitrary disk command to work device.
- **load**: load, but don't run, an application program.
- **work**: set or display the work device and disk numbers.
- **sys**: set or display the system device and disk numbers.

Most of these commands will use other arguments that may be present on the command line. For instance, the **l** and **ls** commands will use an optional filename pattern. Details are given in the Power C manual. The output redirection will work when appropriate, to send output (e.g. a directory listing) to disk or printer.

If the first argument is not one of these built-ins, it is assumed to be the name of a user program. First the work disk is searched for a program file named as the first argument suffixed with ".sh". If it is not found on the work disk, the system disk is searched. If found on one of the two disks, the program is loaded at address $1800, and called by the shell as a subroutine.

This makes Power C a very tailorable environment. If you can express a certain command as a C program, you can easily add it to your available 'vocabulary'. The number of arguments found on the command line is passed to the **main**() routine of the C program as its first argument, with a pointer to an array of character pointers to each of the command line arguments being passed as the second (the traditional *argc* and *argv*). All programs read from standard input and write to standard output by default. Since these can be 'redirected' on the command line, manipulating files is especially easy. Several not-so simple C applications are supplied in source form on the Power C disk, including a text formatter, an object file library editor, a 'filter' for adding page headings to source files in preparation for printing, and a program to search a file for a text pattern.

Power C has another feature to speed things up. Since the shell built-in commands do not use the RAM from $1800 on, a user program that has been loaded there remains untouched by them. Therefore a command program will not even be reloaded from disk if its name is the same as the last user application run (or loaded using the **load** command.) This speeds up the rerunning of a program, and using the **load** command allows standard input data to be on a different disk from the program even on a one-drive setup.

But this can cause some peculiar behaviour. If a program is run a second time without reloading, *global* and *static* data are 'remembered' from the previous run. Thus, C programs originating on other implementations that are 'ported' to Power C often must be modified to explicitly initialize large portions of their global data at run-time. Static data that is local to a function must often be moved outside the function so it can be initialized.

As an aside, note that the easiest way to get around this problem (until the offending program is modified for the Power C environment) is probably by writing the shortest C program:

```
main()
{
    ;
}
```

or, if you prefer, the shortest machine language program, using any assembler package you like, as long as the load address ends up as $1800:

```
rts
```

Name the resultant program "unload", or something similar, and execute it when you want to force a reload of another program.

Memory usage by C programs and the shell

When running, a C program relies on some locations in the shell's memory area, among them:

- $17fa: number of device containing system disk.
- $17fb: number of drive containing system disk.
- $17fc: number of device containing work disk.
- $17fd: number of drive containing work disk.
- $17fe: flag; 0 means standard input is using the default Kernal *chrin* device; 1 means standard input is Kernal logical file #1.
- $17ff: flag; 0 means standard output is using the default Kernal *chrout* device; 1 means standard output is Kernal logical file #2.

It can useful to inspect these locations from C programs. For instance, you may want to suppress prompts if standard input has been redirected. Of course, this will have the disadvantage of making the C source Commodore 64 dependent.

In addition, a few run-time routines frequently used by C programs actually reside in the shell. There is a jump table to these near the beginning of the shell, each entry consisting of a *jmp* instruction to one of the following routines:

- $80e: initialize shell
- $811: *c$1102* code
- $814: *c$funct_init* code
- $817: *printf* code
- $81a: *fprintf* code
- $81d: *sprintf* code
- $820: *c$getc* code (part of *getchar()*)
- $823: *c$2102* code

The *c$funct_init* routine does quick set-up for a C-callable machine-language routine. It is used by the *printf* code, actually. The *c$1102* routine simply pushes the accumulator onto the C run-time stack (the pointer to which is located in $1a,$1b of zero page). The *c$2102* routine just calls *c$1102* to push the value zero onto the C run-time stack.

Finding a free page for the RAM disk interface

The Power C shell occupies or uses nearly all the memory between $801 and $1800. C application programs are loaded at $1800. They set up their run-time stack to grow upwards from the end of the loaded code. This stack is used for automatic variables and some parameter passing, and is pointed to by ($1a,$1b) in the zero-page. Memory dynamically allocated by the C library routines *malloc*, *calloc* and *realloc* is taken from the top-of-memory down. Applications running under the Power C shell potentially use all memory. While the top-of-memory can be set within a single C program by the Power C routine *highmem*, it cannot be set for the entire environment. Standard programs such as *ed*, the excellent Power C text (source-file) editor, and *cc*, the compiler command, while they are not C programs, will certainly try to use all the available memory.

However, by rewriting the shell, it has proven possible to squeeze a free page in there. By carefully compacting the code a little, and re-arranging the use of memory within the shell, the page from $1600 to $16ff (page 22) can be freed.

By loading the RAM disk software and specifying page 22 as the interface page, prior to loading the specially rewritten shell and running it, all C programs will work, and the RAM disk is made accessible as a disk drive.

Other problems (and fixes)

The only hitches occur when the RAM disk does not behave quite like other Commodore disk drives. For instance, a major problem is that the *cc* program, which loads and runs the two passes of the C compiler attempts to concatenate two files by sending the work disk drive the command:

```
c0:file.o=0:xxxtemp1,xxxtemp2
```

Well, the RAM disk does not support concatenation in this fashion. So even though the two passes of the C compiler would load and run properly from the RAM disk and produce correct output on a regular disk drive, the final object file would not be produced correctly if the RAM disk itself was being used as the Power C work disk. But it is also possible to rewrite the *cc* command to perform the concatenation by reading and writing the files. It was while doing this rewrite that I discovered the bug in RAMDOS 3.3 referred to in "On The C Side". Happily that bug has been fixed in the more recent RAMDOS 4.2. RAMDOS releases can be found on various BBSs, and it is good to try and get as recent a version as you can.

In general, though, most programs reading or writing multiple files on the RAM disk are surprisingly slow. I think the process of switching from working with one file to another in the RAM disk software is quite slow. Performance seems to improve noticeably if the input and output is heavily buffered.

A pleasant working environment

After writing the modified shell, and the *cc* command, I combined them with a couple of other support utilities into a compressed archive file, *shellram.arc*. I have uploaded this to the Pro-line Power C BBS (416-276-6811), and I think it may by now be available from other places as well (including the *Transactor* disk for this issue).

So with the modified shell, and the *cc* command, I have a Power C environment in which the editor loads instantly, and takes only a couple of seconds to load or save large source files on the RAM disk. Although the actual computation-bound compilation processes are not much faster, even when the source is on the RAM disk, there are no annoying delays while each pass of the compiler is loaded. In addition, the concatenation of the two temporary files to produce the final object file is a lot faster, and a lot easier on the hardware, both disks and drives. Once compiled, even large programs load in the time it takes to type their name.

In fact, when using Power C with the RAM disk, I often think it is a great tribute to Power C that I was able to tolerate using it with regular disk drives at all.

Inside Geos 128

The info you needed and couldn't find...

by William Coleman

GEOS for the Commodore 128 has been available for quite a while now. A new and improved version (V2.0) has just been released. Unfortunately there still are very few applications written specifically for it. Most are simply C64 GEOS programs that will run in 40-column mode. One of the biggest obstacles has been a lack of information. This article will attempt to remedy this situation by providing all of the information required to program in 80-column mode. There isn't enough room to teach GEOS programming from scratch; therefore, the reader is assumed to already be familiar with GEOS programming on the C64.

Which is which?

Before your application can do its job, it needs to know what computer and what mode it is running under. GEOS (throughout this article the term GEOS is used to signify GEOS 128, unless stated otherwise) provides two variables for the purpose of differentiation.

The first, available to all versions (V1.3 and above), is **c128Flag** ($C013). If bit 7 is set then the application is running under GEOS 128. Note that the 128 DeskTop will parse out programs that can't be run under the current mode or computer (we'll discuss just how a bit later) but the 64 DeskTop can't. If you are writing a 128-only application, it will need to check this flag to make sure that it isn't being run under GEOS 64. Use this routine to check:

```
CheckPuter:
   lda     #$12
   cmp     version
   bpl     10$
   lda     c128Flag
10$:
   rts
```

After calling this routine, BPL will branch if running under a C64.

The second variable, called **graphMode**, is available to GEOS 128 only, all versions. If bit 7 is set then you are in 80-column mode. Virtually all GEOS graphic and text routines check this variable to find out what mode is active.

Graphic doubling

The 80-column screen is, of course, twice as wide as the 40-column screen. GEOS provides a means to automatically double the width and/or X position of all graphics drawn on the screen. The highest three bits of the width/X position byte (or bytes) are used for this as follows:

```
     Bit
15   14   13
0    0    0   Leave X value as is (positive value)
1    1    0   Leave X value as is (negative value)
1    0    n   X = X * 2 + n. Doubled positive.
0    1    n   X = X * 2 - n. Doubled negative.
```

Normally you won't access these bits directly but will instead use the new GEOS constants **ADD1_W**, **ADD1_B**, **DOUBLE_W**, and **DOUBLE_B**. Simply OR the appropriate constant or constants into the width or X position value. Use the byte (B) constants in those areas where a byte (card) value is required, i.e. icon tables. Here's an example:

```
myIconTab:
   .byte   thismany
   .word   mseLeft|DOUBLE_W
   .byte   mseTop

firstIcon:
   .word   myIcon
   .byte   leftEdge|DOUBLE_B
   .byte   yPos
   .byte   width|DOUBLE_B
   .byte   height
   .byte   Service

... etc.
```

The newer versions of *PutChar* allow you to pass negative X position values. In these cases you should EOR the constants rather then ORing them. This will maintain the bit pattern required for negative values.

An astute reader will probably remark that any number when multiplied by two will be even. Therefore, you would not be able to access odd positions (for clearing the screen, etc.). To get around this problem simply OR with **ADD1_B** or **ADD1_W** as appropriate. This will add one to the final result. The only time you will normally need this is when clearing the screen (i.e. 319|**DOUBLE_W**|**ADD1_W**).

The routine used to perform all of this miraculous doubling is called *NormalizeX*. It will be called automatically by all of the GEOS graphic and text routines. In 40-column mode this routine will RTS without doing anything. You can, if you wish, call it from within your application but under normal circumstances (is there really such a thing?) you shouldn't have to. See Table 1 for a more complete description.

Graphics and the 64

While GEOS 128 will correctly handle doubling bits regardless of the video mode, GEOS 64 doesn't know doubling bits from Adam. If you try to use them the graphic routines will try to print way off the screen with obviously lousy results. This makes creating a program that will run in 80-column 128 and GEOS 64 rather difficult.

Probably the easiest way to write a totally compatible program is to keep track of all the places in the program where doubling bits are required, and poke the appropriate values in during initialization.

In those places where the value is loaded into a register, try the following:

```
MyNormalizeX:
   jsr     CheckPuter
   bmi     10$           ; if C128
   lda     $01,x         ; it's a 64
   and     #%00011111    ; so clear the
   sta     $01,x         ; doubling bits
10$:
   rts

Convert:
; pass: r2L-r4 just like in Rectangle
   pha
   txa
   pha
   ldx     #r3
   jsr     MyNormalizeX
   ldx     #r4
   jsr     MyNormalizeX
   pla
   tax
   pla
   rts
```

Now simply use the doubled forms of all X positions and call Convert before all calls to *Rectangle*, *FrameRectangle*, etc. Admittedly, this is a bit of a kludge but it works.

While all of this is a pain for smaller applications, it gets darn near impossible to maintain with larger ones. This is why most large applications have two versions - one for GEOS 64 and one for the 128.

The 128 memory map

The main GEOS 128 bank is bank 1. It is virtually identical to the GEOS 64 memory map with the following exceptions:

1) The I/O area is always banked in.
2) The MMU can be seen at $FF00.
3) Because of #2 the input driver has been moved. The input driver table is still in the same place (in this case, at the *end* of the driver instead of at the beginning).

Under normal circumstances (i.e. when application code is executing), there is no common RAM defined. There is a section of the GEOS Kernal in high memory of bank 0. When GEOS needs to access this, it sets common RAM high and this area will flip in.

The softsprite variables reside in bank 0 between $0400 and $1FFF. When these variables need to be accessed the Kernal will set common RAM low.

GEOS 128 does not swap memory to disk the way the GEOS 64 does. Instead it uses $2000 to $9FFF of bank 0. You can use this area as a buffer but if your application loads Desk Accessories this area will be trashed. There is, unfortunately, no way to prevent this.

While at first glance you would think the 128 has much more room this is not really the case with GEOS 128. The active font must be in front RAM (bank 1), as must the sprite pictures and all tables passed to GEOS routines. While there is a large area in back RAM (bank 0), it is hard to use and will be trashed by DAs.

The 'runnable on' flag

Every GEOS file has a flag in the header block at position OFF128FLAGS. This flag tells the GEOS 128 Kernal what computer and/or mode the program can run under. Only the highest two bits matter; the rest should be zero.

bit 7	bit 6	
0	0	40-column mode only
1	0	40- and 80-column modes
0	1	Does not run under GEOS 128
1	1	80-column mode only

Note that this flag is not recognized by any GEOS 64 version (which is why zero means 40-column only). You can sometimes change this flag with interesting results. Both ***geoAssembler*** and ***geoLinker*** are set so that they will not run on the 128. However, you can change this flag and they will run just fine! In fact, if you don't mind a weird looking screen, you can run them in 80-column mode at twice the normal speed!

Some nasty bugs

There are a couple of serious bugs that you should be aware of. If $1300 is non-zero, the softsprite will not function properly. This one is a real pain to work around. If possible, start your programs at $1300 and make the first instruction **.block 1**. You can then use $0400 to $12ff for buffers.

Since the first version of GEOS there has been a bug in the menu handler that prevents menus from going past position 255. Unfortunately this also applies to the 128 versions *even in 80-column mode!* This one can be quite difficult to work around if you use large menus.

I ran up against this problem when I was writing my 128 terminal. In 40-column mode, the main menu extended to 253. In 80-column mode this translated to about 330 (the BSW80 font is about 1.3 times as wide as BSW40). Of course, it wouldn't work. What I finally did was fool *UseSystemFont* into thinking it was always in 40-column mode so that it would always return BSW40. It works fine like this but the 80-column menu is awfully tiny. Hopefully, V2.0 has fixed these problems, but I'm not holding my breath.

The following tables provide all of the new routines, variables, etc. I would suggest making up a new include file called ***geos128Sym***. You can then include it in all of your 128 programs. If you add everything to ***geosSym*** you will waste assembly time reading variables that aren't used.

Table 1 - *GEOS 128 Routine Descriptions*

DoBOp - $C2EC

Pass: **r0** - BLK1. Pointer to the start of a block of memory.
r1 - BLK2. Pointer to the start of a block of memory.
r2 - Number of bytes to move.
r3L - Bank of BLK1 (0 = bank 0, 1 = bank 1)
r3H - Bank of BLK2 (0 = bank 0, 1 = bank 1)
Y - Mode of operation as follows:

bit 1	bit 2	
0	0	Move from BLK1 to BLK2
0	1	Move from BLK2 to BLK1
1	0	Swap BLK1 with BLK2
1	1	Verify only

Return: **r0-r3** unchanged
X 0 = match, $FF = mismatch (verify only)

Destroy: **a, x, y**

This routine is a low level, general purpose, bank to bank memory move routine. If both source and destination addresses are in the same bank then the source must be greater then the destination. When swapping memory in the same bank the reverse is true.

HideOnlyMouse - $C2F2

Pass: nothing   Return: nothing

Destroy: **a, x, y, r1-r6**

This routine is used to erase the mouse softsprite from the screen when in 80-column mode. It will be redrawn during the next pass through Main Loop. This routine will have no effect if called when in 40-column mode. Usually it is best to call *TempHideMouse*. Since the graphics routines use *TempHideMouse* this routine is worthless if you also call a graphics routine.

MoveBData - $C2E3

Pass: **r0** - Pointer to start of the SOURCE block of memory.
r1 - Pointer to start of DESTINATION block of memory.
r2 - Number of bytes to move.
r3L - SOURCE Bank (0 = bank 0, 1 = bank 1)
r3H - DESTINATION Bank (0 = bank 0, 1 = bank 1)

Return: **r0-r3** unchanged

Destroy: **a, x, y**

This routine simply loads **Y** with zero and calls *DoBOp*. If both areas are in the same bank then DESTINATION must be less than SOURCE.

NormalizeX - $C2E0

Pass: **x** - Pointer to the zero page word to normalize, e.g. LDX #r3.

Return: **x** - unchanged   Destroy: **a**

This routine is used to convert doubling bits to an absolute value. *graphMode* is checked and, if called when in 40-column mode, will simply strip the doubling bits. All GEOS graphics routines call this routine, so normally an application will not need to call it.

SetMouse - $FE89

Pass: nothing   Return: nothing

Destroy: assume **a, x, y, r0-r15**

This routine is called by the interrupt routine just after the keyboard is scanned. It's job is to reset the POT lines. This is only necessary for a mouse; all other drivers will simply

RTS. An application should never need to call this routine directly.

SetMsePic - $C2DA

Pass: **r0** - pointer to 32 bytes of data (two 16 byte images) or ARROW to switch to the default.

Return: nothing   Destroy: **a, x, y, r0-r15**

This routine is used to change the shape of the mouse softsprite (#0) in 80-column mode. Unlike the other seven softsprites, the mouse softsprite is highly optimized so that it will function smoothly as a mouse. It is only 16x8 pixels in size. The first 16 bytes of the table are a mask image. The last 16 bytes are the picture itself. The mouse handler will first AND the area with the mask image. Any 0's in the mask will clear that pixel on the screen. Then the image itself is OR'ed in. Here's an example:

```
TriangleMouse:
   LoadWr0,MSEPIC
   jsr       SetMsePic
   rts

DefaultMouse:
   LoadWr0,ARROW
   jsr       SetMsePic
   rts

; macro to store word in Hi/Lo order

  .macro  rvword  word
    .byte>word,<word
  .endm

MSEPIC:

;mask
     rvword   %0000000000000001
     rvword   %0111111111111011
     rvword   %0111111111101111
     rvword   %0111111110111111
     rvword   %0111111011111111
     rvword   %0111101111111111
     rvword   %0110111111111111
     rvword   %0001111111111111

;image
     rvword   %0000000000000000
     rvword   %0111111111111000
     rvword   %0110000011100000
     rvword   %0110001110000000
     rvword   %0110111000000000
     rvword   %0111100000000000
     rvword   %0110000000000000
     rvword   %0000000000000000
```

SetNewMode - $C2DD

Pass: **graphMode** - set to the NEW mode

Return: nothing

Alters: *rightMargin*, clkreg, calls *UseSystemFont*

Destroy: **a, x, y, r0-r15**

This routine is used to switch graphics mode. The application must redraw the new screen itself. Note that *UseSystemFont* is called; so, if you are using a custom set, you will have to re-enable it. Here's an example:

```
SwitchMode:
; first erase old screen
   jsr    i_Rectangle
   .byte  0,199
   .word  0,319|DOUBLE_W|ADD1_W
; now switch modes
   lda    graphMode
   eor    #%10000000
   sta    graphMode
   jsr    SetNewMode
; now initialize the new screen
   jsr    DrawScreen
   rts
```

SwapBData - $C2E6

Pass: **r0** - BLK1. Pointer to start of a block of memory.
r1 - BLK2. Pointer to start of a block of memory.
r2 - Number of bytes to move.
r3L - Bank of BLK1 (0 = bank 0, 1 = bank 1)
r3H - Bank of BLK2 (0 = bank 0, 1 = bank 1)

Return: **r0-r3** unchanged

Destroy: **a, x, y**

This routine simply loads **Y** with two and calls *DoBOp*. If both areas are in the same bank then BLK2 must be less then BLK1.

TempHideMouse - $C2D7

Pass: nothing   Return: nothing

Destroy: **a, x**

This routine is used to erase all of the softsprites from the screen when in 80-column mode. They will be redrawn during the next pass through Main Loop. This routine will have no effect if called when in 40-column mode. All the GEOS graphics routines call *TempHideMouse* before drawing to the screen. The only time an application will need to call it is when manipulating screen (VDC) memory directly.

VerifyBData - $C2D7

Pass: **r0** - BLK1. Pointer to the start of a block of memory.
r1 - BLK2. Pointer to the start of a block of memory.
r2 - Number of bytes to move.
r3L - Bank of BLK1 (0 = bank 0, 1 = bank 1)
r3H - Bank of BLK2 (0 = bank 0, 1 = bank 1)

Return: **r0-r3** unchanged

Destroy: **a**, **x**, **y**

This routine simply loads **Y** with three and calls *DoBOp*.

TABLE 2

New Equates, Variables and Constants (***geos128Sym***)

```
; Jump Table

  AccessCache     = $c2ef
  ColorCard       = $c2f3
  ColorRectangle  = $c2fb
  DoBOp           = $c2ec
  HideOnlyMouse   = $c2f2
  JmpIndX         = $9d80
  MoveBData       = $c2e3
  NormalizeX      = $c2e0
  SetColorMode    = $c2f5
  SetMsePic       = $c2da
  SetNewMode      = $c2dd
  SwapBData       = $c2e6
  TempHideMouse   = $c2d7
  VerifyBData     = $c2e9

; Variables

  graphMode    = $003f ;bit 7 set = 80 column mode
  scr80Polar   = $88bc    ;copy of VDC reg 24
  scr80Colors  = $88bd    ;copy of VDC reg 26
  vdcClrMode   = $88be    ;current color mode
  keyreg       = $d02f
  clkreg       = $d030
  mmu          = $d500
  VDC          = $d600
  MOUSEBASE    = $fd00
  ENDMOUSE     = $fe80
  config       = $ff00

; Constants

  ADD1_W       = $2000
  ADD1_B       = $20
  ARROW        = 0        ;pass this to SetMsePic
  CIOIN            = $7e
  CKRNLBASIOIN     = $40
  CKRNLIOIN        = $4e
  CRAM64K          = $7f
  DOUBLE_W         = $8000
  DOUBLE_B         = $80
  GR_40            = 0     ;use these two to test
  GR_80            = $80   ;graphMode
  INCOMPATIBLE     = 14    ;new disk error
  INPUT128         = 15    ;new input device
  KEYHELP          = 25
  KEYALT           = 26
  KEYESC           = 27
  KEYNOSCRL        = 7
  KEYENTER         = 11
  OFF128FLAGS      = 96    ;offset into header block
  SCREENBYTEWIDTH  = 80
  SCREENPIXELWIDTH = 640
```

Editorial Note: *The Transactor disk for this issue (Tdisk #27) will include* ***geos128Sym.*** *Also worth noting at this time is the fact that since 9:1 (when I stated that we would support PAL format), all submissions received have been in geoProgrammer format. (Although Francis Kostella went to the trouble of making his article for this issue PAL-compatible.) Consequently, most articles will be in geoProgrammer format using BSW labels and symbols. Generally speaking, an article will be published in the form in which it is received; and that form will most likely be geoProgrammer. - MO*

Loadermaker

Easy GEOS info sectors

by Nicholas J. Vrtis

Once you get the feel for it, coding assembler for GEOS is easy. The routines are pretty easy to access, and you can do some pretty impressive things with just one subroutine call. The problem is that during a development cycle you end up assembling and testing the same program over and over. This process can get messy and inconvenient if you have a program like PRGTOGEOS from *The Official GEOS Programer's Reference Guide.*

The trouble is that once you convert a file to GEOS format, you *must* delete it with GEOS, or you end up with a sector allocated in the BAM but not tied to any program (because the DOS scratch doesn't know about the GEOS info sector). If you are like I am, when you are testing an assembler program (particularly if you are just learning GEOS), you frequently hang the system up and need to reset it. So to put up a new version of the program you need to boot up GEOS, scratch the old program, then boot up your assembler, change the source, reassemble, and reboot GEOS and convert. Finally you get to test your change! With the various copy protection schemes involved in GEOS and your assembler, there are a couple of disk swaps involved in the process.

Loadermaker was written because I am lazy and wanted to shorten the process. All *Loadermaker* does is create a short GEOS program whose sole purpose is to load a non-GEOS program, and start it running in the GEOS environment. Since the loader program doesn't do anything except load the program you are testing, there is not a lot of need to change it. Since the test program is a normal Commodore file, you can use your assembler and scratch, reassemble, etc. to your hearts content without collecting extra GEOS info sectors.

Operation of the program is pretty straightforward. Double click on the icon to get it running. You will get a menu with three options: 'Done' will get you back to the DeskTop, 'Make' starts the process, and 'Help' supplies a little explanation of the program purpose and operation. Selecting 'Make' will present you with a window which asks for the name of the program you want the boot program to load. This should be the name of your assembler output. It does not have to exist at this time. The second question is the name for the boot program which will do the loading. This is the name of the GEOS program you will double click on to get your program loaded and run. If this file already exists, you will be asked if you want it overlaid with the new version or not. If you click on YES, a new copy is written out. If you click on NO, then you will be asked for a new name for the boot loader program. That's it. You will go back to the main menu to either quit or create another boot loader.

The operation of the boot loader program generated by *Loadermaker* is even simpler than *Loadermaker.* All you do is double click on the icon, and you are off and running with your assembler program. The loader runs out of location $7F40, so it will not load a program to that area. It uses the load address found as the first two bytes of the assembler file (à la normal **load "...",8,1**). The starting execution address is assumed to be the same as the starting load address.

Since *Loadermaker* is such a simple GEOS program, it also makes a good sample of how to code one up. While simple, it uses most of the basic GEOS processes (menus, windows, file I/O, etc.). I would recommend two sources of information on GEOS. The first is a shareware manual by Alexander Boyce (2269 Grandview Ave., Apt 1, Cleveland Heights, OH 44106-3144, or check your local BBS). The second is *The Official GEOS Programmers' Reference Guide* by Berkeley (Bantam Computer Books). The latter is a little more complete, but difficult to read and find things in. The former is much easier to read, and better indexed. The *Loadermaker* program is a mixture. The routine names are from Boyce's manual, and the Page Zero definitions are from the Berkeley manual.

Most of the program is a process of setting up Page Zero registers with pointers to a table, calling a GEOS routine to do the work, and then checking the results. I won't try to explain all the options used in each of the tables. Either of the manuals mentioned earlier can do that (and I've included liberal comments in the source). What I *will* do here is point out some of the things which aren't really mentioned in the manuals.

One of the hard parts of GEOS is designing a text screen (believe it or not). Since all the text is proportional, you can't just count characters to decide if they will fit on a line or in a box. I use two different methods. For small areas such as menu options and title lines, I take a guess and then make adjustments after I see it on the screen. Note that when doing menu options, GEOS takes care of the dividing line between all the options, so all you have to do is worry about the width of the box area. If you don't allow enough room, GEOS gets confused and, when you select that entry, it reverse-images more than it should. Other than looking funny, it works for testing. Lining up submenus is really a challenge, since the dividing line is put in by GEOS. Again, take a guess and adjust after you see it on the screen. After a while you can get good enough to count dots on the screen (a screen dot is one pixel).

The second method of figuring out what will fit is the one I use for larger areas (such as the help window text). This is more complicated. I do the text in *geoWrite*, with the margins pulled into the size of the area I am working with. That way, *geoWrite* will tell me what fits, and wrap the lines as needed (for your information, the standard spacing between lines used by *geoWrite* is 10 pixels). There are a lot more 'moving parts' this way, but it doesn't take any longer than if you just tried to guess and adjust each line as you go.

I recommend using bolding for most short text. Plain BSW font (the default font) is a little hard to read. You should also note that if you have a pattern on the screen and put text on it with ten pixels between lines there will be one pixel of pattern showing between most characters, but nine pixels between causes the tops and bottoms of some letters to touch. Therefore, I recommend putting text on a plain background if possible, or line it up with your pattern (patterns are 8 pixels high). I also recommend a space before and after any text which is put on a pattern. Otherwise the first and last characters of the text run into the pattern.

Multiple lines of text in a window are sort of tricky. A carriage return (decimal 13) takes you to the start of the left margin, and the window processor sets the left margin to 0. Any text positioning controls are in reference to the left margin, not the edge of the window. Since the window processor allows multiple text references, I would recommend this method for most windows that need more than one line. It takes five bytes this way (one control, three coordinates, and one 0 at the end).

An alternate method (which I used for the window text) is to use the positioning controls within the text and adjust for where the window will be. This only takes four bytes, but has the disadvantage of needing changes if you move the window.

A word of caution on text: if you forget the 0 at the end of the text, you will usually get a system error and/or hang up GEOS.

It is possible to position some things (click boxes, text, etc.) so they hang outside the window. This tends to mess up your screen, since the window processor only recovers the portion of the screen defined by the window, and it will leave the stuff which hung over on the screen.

One final point about text in GEOS: GEOS uses true ASCII instead of Commodore ASCII. This means that most Commodore assemblers will have trouble assembling anything other than lower case text strings. I modified SYMASS with a new pseudo opcode, .TASC, which tells SYMASS to assemble true ASCII characters instead of Commodore ASCII.

The final challenge now becomes: how do I assemble *Loadermaker* with SYMASS, and make it a GEOS program so I don't have to mess with making GEOS programs for a while again.

Actually, there are three ways. If you have already typed in PRGTOGEOS from the Berkeley Book, then all you need to do is take *Loadermaker*, assemble it, and use a convenient ML monitor to save the object code to disk. Then just run PRGTOGEOS against it. The source for *Loadermaker* is set up with the GEOS info sector on the front where PRGTOGEOS expects it.

If you have not typed in PRGTOGEOS (or don't have the Berkeley Book), then you will need a handy sector editor program to perform some minor disk magic. If you are going to do this, I *strongly* recommend that you start with a disk that doesn't have anything on it that you want (preferably empty). You will be rewriting directory blocks, and it is possible to make a mistake and mess up the disk so that you cannot access anything on it (not likely, but possible).

First, assemble *Loadermaker* and save it out to your scratch disk as a normal Commodore PRG file with a convenient ML monitor. Then you need to use your sector editor to find the directory entry for *Loadermaker*. The directory is at track $12, sector $01. If you started with an empty disk, the *Loadermaker* entry will be the first (and only) entry. It will start with a $82, the starting track and sector, and then the name. You will need to write down the starting track and sector number (this will become the GEOS info sector).

Use your sector editor to read the starting track and sector. The first two bytes of this sector are the track and sector where the *Loadermaker* code starts. Write these numbers down (they will become the program starting sector). The third and fourth bytes tell where the data was saved from (should be $04 $4f). You want to change these four bytes to $00 $FF $03 $15, and write the sector back to disk. Next, you want to reread the directory sector that had the *Loadermaker* entry in it ($12 $01 if the disk was empty). Starting at the $82 in the *Loadermaker* directory entry, you want to change the $82 to $83, then you want to change the next two bytes to be the track and sector of the program starting sector you wrote down from before. The name portion of a directory entry is always the same length, and the name is padded with shifted spaces ($A0's). Immediately following the name, you want to enter the track and sector numbers of the info sector you recorded previously. Following the info sector, you want to enter the following five bytes: $00 $06 $58 $01 $01. Finally, rewrite the directory sector back to disk.

A third method is the one I use after I have tested a new program and want to make it into a normal GEOS program. I just 'steal' an info sector from another GEOS program. Again, I urge doing this on an empty disk - just in case. What I do is copy a GEOS program that has an info sector similar to what I want. Then I point to that sector as the info sector of my new program, and fix up the load address, etc. Finally, I boot up GEOS and trash the copy I took the info sector from. I then use GEOS to validate the disk (because it has scratched my info sector). While this sounds sort of complicated, I don't have to do it enough to make it worth my while to bother keying in PRGTOGEOS, and having to key in the info sector for each new program.

Loadermaker.src

```
BF  100 sys 49152
OL  110 ;"LOADERMAKER.SRC -- creates a ml loader program on GEOS disk"
DG  120 ;"Nick Vrtis     -- January 1988"
IP  130 ;
CB  140 ; work registers as in geos programmers reference
MK  150 r0  = $02
PL  160 r1  = $04
JP  170 r4  = $0a
FB  180 r6  = $0e
HP  190 r9  = $14
EC  200 r10 = $16
CC  210 a0  = $fb
FD  220 a1  = $fd
MF  230 ;
AD  240 .tasc              ;assemble true ascii literals
AH  250 ;
DB  260 * = $5000-252      ;allow for geos info sector ($4f04)
MC  270 .byte $bf
LP  280 .byte $00,$00,$00,$7f,$ff,$fe,$40,$1f
GK  290 .byte $fe,$5f,$80,$7e,$6f,$ff,$7e,$77
NJ  300 .byte $fe,$7e,$77,$05,$fe,$76,$03,$fe
ON  310 .byte $74,$ff,$fe,$75,$80,$7e,$75,$ff
FN  320 .byte $fe,$75,$80,$7e,$74,$ff,$e6,$76
PJ  330 .byte $03,$1a,$77,$07,$fc,$77,$ff,$fc
PO  340 .byte $77,$a3,$fc,$77,$b8,$fa,$70,$3f
JG  350 .byte $06,$7f,$ff,$fe,$00,$00,$00
JL  360 .byte $83,$06,$00
LA  370 .word begin        ;starting load address
PI  380 .word ldrend       ;ending load address
BJ  390 .word begin        ;starting execution address
PB  400 .asc "LoaderMakerV1.0"
OB  410 .byte 0,0,0,0,0
KJ  420 .asc "Nick Vrtis -- 1988"
CI  430 .byte 0,0
EJ  440 .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
OE  450 .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
NA  460 .asc "Create a loader program on GEOS disk to load ML files "
OC  470 .asc "created by a non-GEOS assembler"
EN  480 .byte 0
AG  490 ;
HN  500 * = $5000
CF  510 moveto =$7f40 ;where boot code will execute from
JC  520 begin lda #<title
KG  530 sta r0
EO  540 lda #>title
AE  550 sta r0+1
KF  560 jsr grphic     ;do the opening credits
AL  570 ;
CG  580 doagain = *    ;restart point
CA  590 lda #<mainmenu
AL  600 sta r0
CB  610 lda #>mainmenu
GI  620 sta r0+1
NN  630 lda #1         ;cursor on option #2
AF  640 jmp menu
AA  650 ;
HE  660 doquit = *     ;handle the quit option
AK  670 jmp restrt     ;back to desktop
OB  680 ;
BP  690 dohelp  = *    ;handle the help option
GO  700 ldx #<helpw
AP  710 ldy #>helpw
HP  720 jsr xywindow
OG  730 jmp drwmnu     ;redraw the menu
KF  740 ;
JA  750 domake  = *    ;handle the make option
KP  760 jsr cmenus     ;close all menus
EC  770 lda #<pgmname
GP  780 sta r10
ED  790 lda #>pgmname
DO  800 sta r10+1
PL  810 ldx #<loadnw
JM  820 ldy #>loadnw
FG  830 jsr xywindow
JJ  840 lda r0         ;check for cancel
GI  850 cmp #2
DB  860 beq doagain    ;..yes-restart
MN  870 ;
GG  880 getln    = *   ;get name for loader file
JI  890 lda #<ldrname
OG  900 sta r10
JJ  910 lda #>ldrname
LF  920 sta r10+1
GN  930 ldx #<ldrnw
AO  940 ldy #>ldrnw
NN  950 jsr xywindow
BB  960 lda r0         ;check for cancel
OP  970 cmp #2
LI  980 beq doagain    ;..yes-restart
NO  990 lda #<ldrname
ME  1000 sta r6
NP  1010 lda #>ldrname
CC  1020 sta r6+1
PO  1030 lda drvsrch  ;check only 'active' drive for new name
OA  1040 pha          ;save current setting
FA  1050 lda #0
AL  1060 sta drvsrch
PK  1070 jsr lookup   ;try to find loader name
AM  1080 pla
BB  1090 sta drvsrch  ;reset drive search flag
BM  1100 txa          ;check return from lookup
JC  1110 bne fileok   ;..not found--good
EF  1120 ldx #<erafnw
OF  1130 ldy #>erafnw
LJ  1140 jsr xywindow
DE  1150 lda r0       ;see if ok to overlay
AM  1160 cmp #4
FB  1170 beq getln    ;..no-get a new loader name
LK  1180 lda #<ldrname
OP  1190 sta r0
LL  1200 lda #>ldrname
EN  1210 sta r0+1
IH  1220 jsr delete   ;erase current version
EE  1230 ;
KN  1240 fileok = *   ;file setup is ok
PI  1250 ldx #0       ;move loader code to where it will run from
DK  1260 ldrmove = *
CO  1270 lda ldrcode,x
LK  1280 sta moveto,x
GO  1290 inx
FK  1300 cpx #<ldrend-ldrcode
IG  1310 bne ldrmove
OH  1320 clc          ;calc end address of save
FF  1330 lda #<ldrend-ldrcode
OA  1340 adc #<moveto
IO  1350 sta savend
PG  1360 lda #>ldrend-ldrcode
IC  1370 adc #>moveto
PN  1380 sta savend+1
JI  1390 lda #0        ;try starting on 1st page of directory
CG  1400 sta r10
MJ  1410 lda #<ldrinfo
GP  1420 sta r9
```

```
MK 1430 lda #>ldrinfo
MM 1440 sta r9+1
NJ 1450 jsr save      ;save new file
BM 1460 jmp doagain   ;redo menu/options
ED 1470 ;
FE 1480 xywindow = *  ;produce window pointed to by x/y regs
GI 1490 stx r0
GF 1500 sty r0+1
ME 1510 jmp window       ;got here via jsr/return directly
GG 1520 ;
IG 1530 mainmenu .byte 0 ;start @ top of screen
DG 1540 .byte 14         ;to +14 down
GD 1550 .word 0          ;left edge
FF 1560 .word 92         ;right edge
MA 1570 .byte 3          ;3 horizontal options
AO 1580 .word done
KF 1590 .byte $80        ;submenu option
FL 1600 .word donemenu   ;submenu defination
NH 1610 .word make
NF 1620 .byte 0          ;flash & do option
CB 1630 .word domake
MK 1640 .word help
GG 1650 .byte 0
DE 1660 .word dohelp
FH 1670 done .asc "Done" : .byte 0
AJ 1680 make .asc "Make" : .byte 0
NN 1690 help .asc "Help" : .byte 0
KB 1700 ;
IP 1710 donemenu .byte 14  ;start below main
NL 1720 .byte 14+14        ;still 14 pixels high
KO 1730 .word 0            ;left edge
AB 1740 .word 29           ;right edge
OH 1750 .byte $80+1        ;one vertical entry
OF 1760 .word quit
DF 1770 .byte 0            ;flash & do
FP 1780 .word doquit
LK 1790 quit .asc "Quit" : .byte 0
OH 1800 ;
AH 1810 title .byte $05,28              ;fill pattern
AD 1820 .byte $01,0,0,0                 ;top/left corner
NN 1830 .byte $03:.word 320:.byte 199 ;fill whole screen
IC 1840 .byte $05,9                     ;new fill pattern
BL 1850 .byte $03:.word 320:.byte  20 ;fill top title line
CK 1860 .byte $06:.word 8:.byte 190    ;output title
OM 1870 .byte $18,$20,$1a               ; bold+outlined
CH 1880 .asc "LoaderMaker"
BH 1890 .byte $1b,$18                   ; just bold
PO 1900 .asc " V1.0 Machine Language Loader Maker "
JA 1910 .byte $16,8,0,142               ; new position for text
ME 1920 .asc " Nicholas J. Vrtis "
JP 1930 .byte $16,200,0,142
FB 1940 .asc " Copyright 1988 "
PD 1950 .byte $16,8,0,158
EI 1960 .asc " 5863 Pinetree S.E. "
HF 1970 .byte $16,8,0,174
JJ 1980 .asc " Kentwood, MI 49508 "
FF 1990 .byte $1b,0        ;back to plaintext at end
GE 2000 ;
LE 2010 helpw .byte $01    ;non-standard sized window here
KE 2020 .byte 22,178       ;top/bottom pixels
OB 2030 .word 4            ;left edge
NJ 2040 .word 305          ;right edge
NA 2050 .byte $0b,2,15
PP 2060 .word helpmsg
AB 2070 .byte $01,31,135,0 ;-cancel- box
LL 2080 helpmsg .byte $18
JI 2090 .asc "This program is used to create a GEOS program which"
AN 2100 .byte $16,6,0,47   ;change here is window size adjusted
DF 2110 .asc "can be run from the DeskTop.  The GEOS program will"
PL 2120 .byte $16,6,0,57
DH 2130 .asc "then load and run an assembler program created as a"
GN 2140 .byte $16,6,0,67
HJ 2150 .asc "'normal' Commodore program.  This avoids having to"
NO 2160 .byte $16,6,0,77
LB 2170 .asc "run a separate program to 'convert' your assembler"
EA 2180 .byte $16,6,0,87
ND 2190 .asc "program each time you reassemble it.  A 'normal'"
LB 2200 .byte $16,6,0,97
PF 2210 .asc "Commodore program has the load address as the first"
CD 2220 .byte $16,6,0,107
AH 2230 .asc "two data bytes of the file, and starts execution at"
KE 2240 .byte $16,6,0,117
EO 2250 .asc "that address after being loaded."
CJ 2260 .byte $1b,0
EF 2270 ;
GI 2280 ldrnw .byte $81    ;standard size window
BJ 2290 .byte $0b,10,30
IP 2300 .word lnmsg
LJ 2310 .byte $0b,10,40    ;two lines of text in this window
CK 2320 .word lnmsg2
LE 2330 .byte $0d,10,60    ;get input in window
BC 2340 .byte <r10,16      ;r10 is buffer pointer/max 16 characters input
GF 2350 .byte $02,17,78    ;-cancel- box
MC 2360 .byte 0
MG 2370 lnmsg .byte $18
CD 2380 .asc "Enter name for NEW loader"
KE 2390 .byte 0
HF 2400 lnmsg2 .asc "program to be created."
IC 2410 .byte $1b,0
KO 2420 ;
PA 2430 loadnw .byte $81   ;again a standard sized window
HC 2440 .byte $0b,10,30
AJ 2450 .word plmsg
DE 2460 .byte $0d,10,60
KH 2470 .byte <r10,16
JF 2480 .byte $02,17,78    ;cancel box
OK 2490 .byte 0
AP 2500 plmsg .byte $18
OB 2510 .asc "Enter name of PROGRAM to load."
GJ 2520 .byte $1b,0
IF 2530 ;
PE 2540 erafnw .byte $81
FJ 2550 .byte $0b,10,30
FP 2560 .word errmsg
LK 2570 .byte $0b,10,40
BN 2580 .word errmsg2
AL 2590 .byte $03,17,60     ;-yes- box
JK 2600 .byte $04,17,78     ;-no-  box
GC 2610 .byte 0
GL 2620 errmsg .byte $18
GN 2630 .asc "That file already exists."
EE 2640 .byte 0
IO 2650 errmsg2 .asc "OK to overlay ?"
CC 2660 .byte $1b,0
EO 2670 ;
PL 2680 ldrinfo .word ldrname
EB 2690 .byte $03,$15,$bf
MD 2700 .byte $ff,$ff,$ff,$80,$00,$01,$aa,$a0
OA 2710 .byte $01,$a0,$55,$81,$90,$00,$01,$88
EI 2720 .byte $01,$81,$80,$f8,$01,$89,$fc,$01
AC 2730 .byte $83,$00,$01,$8a,$7f,$81,$82,$00
KD 2740 .byte $01,$8a,$7f,$81,$83,$00,$09,$89
KM 2750 .byte $fc,$a5,$80,$f8,$03,$88,$00,$01
FC 2760 .byte $80,$54,$03,$88,$05,$05,$8d,$40
CI 2770 .byte $a9,$80,$00,$01,$ff,$ff,$ff
NC 2780 .byte $83,$06,$00
BM 2790 .word moveto         ;begin address of save
MJ 2800 savend *=*+2         ;end address for save
LO 2810 .word moveto+17      ;execution start address
DJ 2820 .asc "LoaderMakerV1.0"
CJ 2830 .byte 0,0,0,0,0
OA 2840 .asc "Nick Vrtis -- 1988"
GP 2850 .byte 0,0
IA 2860 .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
CM 2870 .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
MJ 2880 .asc "Load and Run a GEOS program created by a non-GEOS assembler.  "
OP 2890 .asc "This program loads "
KM 2900 ;
```

```
HI 2910 ldrname .byte 0
BD 2920 *=*+16               ;allow for full name + trailing null
IO 2930 ;
OP 2940 ldrcode = *          ;loader code starts here
MK 2950 pgmname .byte 0
MF 2960 *=*+16
AB 2970 ;
PK 2980 loadml lda #<moveto  ;address of name of program to load
CB 2990 sta r6
EJ 3000 lda #>moveto
IO 3010 sta r6+1
JJ 3020 jsr lookup
KL 3030 txa                  ;check return
BK 3040 beq diskok           ;..found it ok
KL 3050 jmp restrt           ;else abandon the load & go back to desktop
KG 3060 ;
NN 3070 diskok lda dentry+2  ;sector of file beginning
EC 3080 sta r1+1
MN 3090 lda dentry+1         ;track of file beginning
GH 3100 sta r1
AB 3110 lda #<buf0           ;set pointer to buffer for read
AJ 3120 sta r4
HO 3130 lda #>buf0
GG 3140 sta r4+1
HH 3150 jsr read             ;read 1st buffer
BF 3160 lda buf0+2           ;get program load adress
JK 3170 sta a0
MH 3180 sta a1               ;also store as starting address
MG 3190 lda buf0+3
JI 3200 sta a0+1
FJ 3210 sta a1+1
DJ 3220 ldx #3               ;skip t/s & address
NO 3230 bne skipread         ;..unconditional-skip 1st read
AD 3240 readpgm lda buf0     ;get track/sector of next block
MA 3250 sta r1
OK 3260 lda buf0+1
CO 3270 sta r1+1
MC 3280 jsr read
NK 3290 ldx #1               ;skip t/s pointer for all other blocks
IN 3300 skipread ldy #$ff    ;assume a full sector
MG 3310 lda buf0             ;check next track pointer
KC 3320 bne goodnxtt         ;..good next track
KJ 3330 ldy buf0+1           ;else this is last sector/pick up # valid bytes
HD 3340 goodnxtt sty r0      ;save # valid bytes (255 for full sector)
BG 3350 ldy #0
BD 3360 moveloop inx
KM 3370 lda buf0,x           ;get input byte
CC 3380 sta (a0),y           ;store it
PA 3390 inc a0               ;bump pointer
EI 3400 bne a0ok
ND 3410 inc a0+1
HD 3420 a0ok cpx r0          ;check if to end of buffer
CE 3430 bne moveloop         ;..not yet/more to do
PP 3440 lda buf0             ;check if more sectors to go
CP 3450 bne readpgm          ;..yes (next track pointer is non-zero)
HA 3460 jmp (a1)             ;now go start program
EA 3470 ;
KP 3480 ldrend = *           ;end of loader code to move
IB 3490 ;
MI 3500 buf0    = $8000      ;geos buffer 0
KG 3510 dentry  = $8400      ;directory entry from lookup
PP 3520 drvsrch = $886e      ;drive search flag
AE 3530 ;
JB 3540 grphic = $c136       ;graphic table processor
AH 3550 menu   = $c151       ;menu processor
EB 3560 drwmnu = $c193       ;redraw the menu
PN 3570 cmenus = $c1bd       ;close all menus
LE 3580 read   = $c1e4       ;read a sector
OD 3590 save   = $c1ed       ;save a geos file
IF 3600 lookup = $c20b       ;lookup file in directory
PK 3610 restrt = $c22c       ;reload desktop & restart geos
MD 3620 delete = $c238       ;delete geos file
KC 3630 window = $c256       ;window processor routine
OK 3640 ;
OB 3650 .end
```

The lines below should be entered (while the lines above are still in memory) if you are NOT using Nick Vrtis' modified version of the Symass assembler from the Transactor disk for this issue. They contain the ASCII bytes for the messages in the Loadermaker program.

```
FD  100 sys 700
GG  240 ;
BB  400 .byte  76,111, 97,100,101,114, 77, 97
BD  401 .byte 107,101,114, 86, 49, 46, 48
HF  420 .byte 78,105,99,107,32,86,114,116
FJ  421 .byte 105,115,32,45,45,32,49,57,56
BI  422 .byte 56
DG  460 .byte 67,114,101,97,116,101,32,97
DI  461 .byte 32,108,111,97,100,101,114,32
AC  462 .byte 112,114,111,103,114,97,109
NL  463 .byte 32,111,110,32,71,69,79,83,32
EP  464 .byte 100,105,115,107,32,116,111
AE  465 .byte 32,108,111,97,100,32,77,76
MG  466 .byte 32,102,105,108,101,115,32
AL  470 .byte 99,114,101,97,116,101,100,32
NE  471 .byte 98,121,32,97,32,110,111,110
LN  472 .byte 45,71,69,79,83,32,97,115,115
JF  473 .byte 101,109,98,108,101,114
IL 1670 done .byte 68,111,110,101
LH 1671 .byte 0
AE 1680 make .byte 77,97,107,101
FI 1681 .byte 0
DB 1690 help .byte 72,101,108,112
PI 1691 .byte 0
PM 1790 quit .byte 81,117,105,116
DP 1791 .byte 0
JN 1880 .byte 76,111,97,100,101,114,77,97
GA 1881 .byte 107,101,114
GC 1900 .byte 32,86,49,46,48,32,77,97,99
PD 1901 .byte 104,105,110,101,32,76,97
AP 1902 .byte 110,103,117,97,103,101,32
KN 1903 .byte 76,111,97,100,101,114,32,77
AF 1904 .byte 97,107,101,114,32
EA 1920 .byte 32,78,105,99,104,111,108,97
EP 1921 .byte 115,32,74,46,32,86,114,116
LA 1922 .byte 105,115,32
MN 1940 .byte 32,67,111,112,121,114,105
LO 1941 .byte 103,104,116,32,49,57,56,56
LF 1942 .byte 32
GI 1960 .byte 32,53,56,54,51,32,80,105
JK 1961 .byte 110,101,116,114,101,101,32
DF 1962 .byte 83,46,69,46,32
KA 1980 .byte 32,75,101,110,116,119,111
ND 1981 .byte 111,100,44,32,77,73,32,52
HG 1982 .byte 57,53,48,56,32
LL 2090 .byte 84,104,105,115,32,112,114
BM 2091 .byte 111,103,114,97,109,32,105
PK 2092 .byte 115,32,117,115,101,100,32
KA 2093 .byte 116,111,32,99,114,101,97
CP 2094 .byte 116,101,32,97,32,71,69,79
GH 2095 .byte 83,32,112,114,111,103,114
LE 2096 .byte 97,109,32,119,104,105,99
MB 2097 .byte 104
EL 2110 .byte 99,97,110,32,98,101,32,114
HI 2111 .byte 117,110,32,102,114,111,109
GC 2112 .byte 32,116,104,101,32,68,101
AL 2113 .byte 115,107,84,111,112,46,32,32
KA 2114 .byte 84,104,101,32,71,69,79,83
FO 2115 .byte 32,112,114,111,103,114,97
AO 2116 .byte 109,32,119,105,108,108
JH 2130 .byte 116,104,101,110,32,108,111
BL 2131 .byte 97,100,32,97,110,100,32,114
OL 2132 .byte 117,110,32,97,110,32,97,115
JL 2133 .byte 115,101,109,98,108,101,114
IP 2134 .byte 32,112,114,111,103,114,97
ID 2135 .byte 109,32,99,114,101,97,116
PN 2136 .byte 101,100,32,97,115,32,97
MC 2150 .byte 39,110,111,114,109,97,108
```

```
OE 2151 .byte 39,32,67,111,109,109,111
BI 2152 .byte 100,111,114,101,32,112,114
GP 2153 .byte 111,103,114,97,109,46,32,32
DI 2154 .byte 84,104,105,115,32,97,118
AO 2155 .byte 111,105,100,115,32,104,97
DJ 2156 .byte 118,105,110,103,32,116,111
EF 2170 .byte 114,117,110,32,97,32,115
AB 2171 .byte 101,112,97,114,97,116,101
OB 2172 .byte 32,112,114,111,103,114,97
CA 2173 .byte 109,32,116,111,32,39,99,111
BP 2174 .byte 110,118,101,114,116,39,32
GP 2175 .byte 121,111,117,114,32,97,115
EO 2176 .byte 115,101,109,98,108,101,114
AO 2190 .byte 112,114,111,103,114,97,109
LP 2191 .byte 32,101,97,99,104,32,116,105
HA 2192 .byte 109,101,32,121,111,117,32
JA 2193 .byte 114,101,97,115,115,101,109
AC 2194 .byte 98,108,101,32,105,116,46,32
FO 2195 .byte 32,65,32,39,110,111,114,109
KL 2196 .byte 97,108,39
FB 2210 .byte 67,111,109,109,111,100,111
AN 2211 .byte 114,101,32,112,114,111,103
FD 2212 .byte 114,97,109,32,104,97,115,32
AD 2213 .byte 116,104,101,32,108,111,97
ON 2214 .byte 100,32,97,100,100,114,101
FI 2215 .byte 115,115,32,97,115,32,116
GO 2216 .byte 104,101,32,102,105,114,115
KJ 2217 .byte 116
OC 2230 .byte 116,119,111,32,100,97,116
DC 2231 .byte 97,32,98,121,116,101,115,32
LC 2232 .byte 111,102,32,116,104,101,32
EI 2233 .byte 102,105,108,101,44,32,97
JE 2234 .byte 110,100,32,115,116,97,114
KD 2235 .byte 116,115,32,101,120,101,99
OC 2236 .byte 117,116,105,111,110,32,97
OK 2237 .byte 116
JM 2250 .byte 116,104,97,116,32,97,100
FD 2251 .byte 100,114,101,115,115,32,97
MC 2252 .byte 102,116,101,114,32,98,101
FF 2253 .byte 105,110,103,32,108,111,97
ID 2254 .byte 100,101,100,46
ON 2380 .byte 69,110,116,101,114,32,110
AN 2381 .byte 97,109,101,32,102,111,114
HC 2382 .byte 32,78,69,87,32,108,111,97
FP 2383 .byte 100,101,114
AO 2400 lnmsg2 .byte 112,114,111,103,114
EP 2401 .byte 97,109,32,116,111,32,98,101
JD 2402 .byte 32,99,114,101,97,116,101
GB 2403 .byte 100,46
AG 2510 .byte 69,110,116,101,114,32,110
OE 2511 .byte 97,109,101,32,111,102,32,80
BA 2512 .byte 82,79,71,82,65,77,32,116
NL 2513 .byte 111,32,108,111,97,100,46
ND 2630 .byte 84,104,97,116,32,102,105
KK 2631 .byte 108,101,32,97,108,114,101
KK 2632 .byte 97,100,121,32,101,120,105
OM 2633 .byte 115,116,115,46
OG 2650 errmsg2 .byte 79,75,32,116,111,32
HJ 2651 .byte 111,118,101,114,108,97,121
CF 2652 .byte 32,63
FI 2820 .byte 76,111,97,100,101,114,77,97
FK 2821 .byte 107,101,114,86,49,46,48
LM 2840 .byte 78,105,99,107,32,86,114,116
IL 2841 .byte 105,115,32,45,45,32,49,57
NC 2842 .byte 56,56
KL 2880 .byte 76,111,97,100,32,97,110,100
PL 2881 .byte 32,82,117,110,32,97,32,71
DM 2882 .byte 69,79,83,32,112,114,111,103
AP 2883 .byte 114,97,109,32,99,114,101,97
JK 2884 .byte 116,101,100,32,98,121,32,97
DK 2885 .byte 32,110,111,110,45,71,69,79
HM 2886 .byte 83,32,97,115,115,101,109,98
AD 2887 .byte 108,101,114,46,32,32
LN 2890 .byte 84,104,105,115,32,112,114
KO 2891 .byte 111,103,114,97,109,32,108
CB 2892 .byte 111,97,100,115,32
```

An Introduction To GEOS Files

Using the high-level disk routines

by Francis G. Kostella

While coding my first attempts at a working GEOS program, I was overwhelmed - and more than a little confused - at the seeming complexity of the GEOS kernal. After all, there are almost 200 callable routines and hundreds of variables available to the programmer, and I believe I've had the opportunity to use most of them the wrong way! But a bit of persistence does pay off and, after a while, the structure begins to make sense. What once seemed like hoops to be jumped through became simple methods of achieving complex results.

My first few programs avoided using any type of disk access, except to exit the program and reload the DeskTop. But later I became a bit bolder and decided to take a look at the dozens of disk routines available and try to use them in my programs (besides, my programs were starting to eat up all the available memory and I'd have to load and save sections to disk). A first glance in *The Official GEOS Programmers Reference Manual* from Berkeley Softworks showed over 50 disk routines available, so I prepared myself to write a number of complex setup routines and to do hours of debugging.

Well, my first efforts failed miserably, but further study showed that I was using too many routines. Looking back it seems I was trying to use as many of the available routines as possible, when one or two would have sufficed. The great majority of the disk calls are the primitives that make up the higher level routines that load and save VLIR and GEOS SEQ files.

Loading a file is easily accomplished once you have the filename, and the record number if the file is a VLIR file. To load a GEOS SEQuential structure file, we simply put the address of the null-terminated filename string into the zero page register **r6** ($0E/0F), store a zero in **r0** ($02) and then JSR to $C208 [*GetFile* (LOAD)]. (Throughout this article, and in the program listing, the hex address of the routine is used, followed by the BSW label and the Boyce label.) The file is loaded to the address in the file's Header Block. If we want to load the file to a different address, we store a 1 in **r0**, and pass the address to load in **r7** ($10/11). Filenames are easily obtained from disk or from the user with a dialog box (see the example program).

If we're trying to load a record from a VLIR file we first open the file by passing the pointer to the filename in **r0** ($02/03) then calling $C274 [*OpenRecordFile* (VOPEN).] Next we pass the record number we want in .A and JSR $C280 [*PointRecord* (GOTO).] We're now pointing at the record, to load it pass the load address in **r7**, and the number of bytes expected in **r2** ($06/07). Finally, after the record is read in, (assuming we're done reading records) we close the file with JSR $C277 [*CloseRecordFile* (VCLOSE).]

To save a file of any type, we need to create an area in memory that will be saved to disk as the Header Block attached to the file. The Header is 256 bytes long and will tell the save routine the file type, file structure, load address, etc. Exact descriptions of the structure of the Header can be found in either Alex Boyce's manual or the BSW PRG, but there are a few bytes we'll use when saving a file that should be explained.

When the Header is written to disk, the first two bytes will be replaced with $00 $FF to indicate that the entire sector is used and no other, but while the Header is still in RAM, these two bytes will be used to point to a null-terminated filename. This is the name that will be used in the file's directory entry. A few other bytes will be used to build the directory entry. Byte 68 in the Header will hold the normal DOS file type (usually $83, USR). Byte 69 will describe one of the GEOS file types (font, application data, desk accessory, and so on). Byte 70 will tell us the GEOS file structure, 1 for VLIR and 0 for GEOS SEQUENTIAL. The next two pairs of bytes are addresses: 71/72 is the load/beginning address, 73/74 is the end address.

To save an area of memory as a GEOS SEQUENTIAL file, we set up the above mentioned bytes and addresses, load the address of the Header into **r9** ($14/15), put a zero into **r10l** ($16) and call $C1ED [*SaveFile* (SAVE).]

To save a VLIR record, we must first create the file. Call $C1ED as above, but set the beginning address in the Header to $0000 and the end address to $FFFF. (If the VLIR file is an executable file, put the load address of the first record into Header bytes 71/72, the load address-1 into 73/74, and the JMP address into 75/76.) Now we have to create the records, first open it, call $C289 126 times, then close the VLIR file, like this:

```
    lda #<filnam
    sta r0  ; ($02)
    lda #>filnam
    sta r0+1
    jsr $c274 ; openrecordfile (vopen)
    ldy #126
  aloop
    jsr $c289 ; appendrecord (append)
    dey
    bne aloop
    jsr $c277 ; closerecordfile (vclose)
```

To write a record, select the record by calling $C280 [*PointRecord* (GOTO)] with the record number in .A, put the beginning save address in **r7** ($10/11) and the number of bytes to save in **r2** ($06/07) and JSR $C28F [*WriteRecord* (VSAVE).]

The GEOS disk routines do their own error checking and usually return error numbers in the .X register. Using the above routines for simple loading and saving, the only error I've had to deal with is #11: record empty, when attempting to to read a nonexistent VLIR record. Of course, more complex applications will want more elaborate error checking. See the above-mentioned programmer's reference guides for more details.

Another thing I should mention, is that the above routines assume that you are not changing disk devices nor swapping disks. Changing drives is accomplished by calling $C2B0 [*SetDevice* (DRVSET)] with the device number in .A. This should be followed by a call to $C2A1 [*OpenDisk* (OPNDSK)] which is also used to read a newly inserted disk. These two routines, when used, should be called before using any of the load or save routines detailed above.

About the programs

Program 2, *Icon Definer* will translate a photo from a Photo Album into a form readily digestible by your assembler. I use this program to 'grab' my compacted icon drawings from *geoPaint* for use in my programs. In fact, the icons used in the program were translated by an earlier version of this program.

A few notes: The first three bytes of the graphic are the width and height of the bitmap; you should delete them or comment them out as they will interfere with the GEOS uncompacting routines. Also, the last few bytes are the colour information for the bitmap. If you have a copy of the BSW PRG, see Appendix D; otherwise, you can leave them in, or calculate them from the width and height bytes.

The resulting SEQ file can be converted to a PRG file with Program 1, which is from an earlier *Transactor*. If you're translating lots of icons, use the DOS command C0: to concatenate a group of SEQ files to save time. If you'd like to have a GEOS Header attached to the output file, change byte 69 to 3. The program is easily modified to accept different input and output files, so experiment. I'd like to thank Joe Buckley for explaining to me the use of the undocumented routine *AppendRecord*.

Program 1: *"convertseqtopal"*

```
CM  0 rem save"convertseqtopal",8
PI  10 open1,8,2,"0:filename":l=1000
EA  20 print"{clr}{down}":fori=0to8:printl;:l=l+2
DP  30 get#1,a$:s=-(st=0):ifs=0goto50
BG  40 ifa$<>chr$(13)thenprinta$;:goto30
DG  50 print:nexti
MB  60 ifsthenprint"l=";l;":poke152,1:goto20"
KN  70 fori=631to639+s:pokei,13:next
KH  80 poke198,9+s:close1-s:print"{home}";:end
OM  90 :
```

Program 2: *"icon definer.src"*

```
LB 1000 open2,8,1,"0:id"
DM 1010 sys700
NA 1020 .opt o2
MH 1030 ;
ME 1040 start  =$0304
FF 1050 ;--equates--
KD 1060 r0      =$02
NE 1070 r1      =$04
AG 1080 r2      =$06
KJ 1090 r5      =$0c
DH 1100 r7      =$10
PI 1110 r9      =$14
ML 1120 r10     =$16
MG 1130 a2      =$70
PH 1140 a3      =$72
CJ 1150 a4      =$74
PD 1160 disbuf =$2f      ; fore/back screen write (displaybufferon)
IA 1170 ;
IF 1180 pload =$0b00     ; load photo here, is also width byte
OG 1190 pdepth =$0b01    ; depth stored here
AP 1200 pstart =$0b03    ; 1st byte of bitmap
EI 1210 sbegin =$20d0    ; start seq save here
MG 1220 ; max 349 lines of 16 bytes each
FG 1230 recvec = $84b1   ; recover screen from dialog box
OE 1240 ;
IL 1250 *= start
CG 1260 ;
CM 1270 ;1st 4 bytes commented out
KL 1280 ; they will be placed in the
LB 1290 ; file header by "maketogeos"
CP 1300 ;.byte $00,$ff
AJ 1310 ;.byte 3,21 ; 3x21 icon
DP 1320 .byte $bf,$ff,$ff,$ff,$82,$20,$01
BA 1330 .byte $84,$50,$01,$89,$88,$01
AP 1340 .byte $84,$84,$01,$8e,$52,$01,$87
PA 1350 .byte $31,$01,$83,$98,$81,$81,$c1,$81,$f4
IJ 1360 .byte $e3,$01,$8c,$ff,$01,$f4,$a1,$01
BD 1370 .byte $84,$52,$01,$84,$52,$c1,$84,$a1
MF 1380 .byte $79,$84,$80,$55,$8f,$a0,$6d,$84
BG 1390 .byte $a1,$57,$80,$52,$cb,$80,$3c,$0d
KO 1400 .byte $ff,$ff,$ff
IP 1410 ;
IF 1420 .byte $83        ;c= filetype user
GN 1430 .byte 6          ;application
CC 1440 .byte 0          ;geos seq file
GG 1450 .word saddr      ;start addr
OB 1460 .word endcod     ;end addr
JN 1470 .word stjump     ;start addr jump
OD 1480 ;
LI 1490 .asc "icon definerv1.0"
IH 1500 .byte 0,0,0,0
EG 1510 .asc "f.g.kostella"
MI 1520 .byte 0,0,0,0
AH 1530 ;
IA 1540 ;the rest of the header is not used here
EI 1550 ;
```

```
ON 1560 ;---start geos file----
IM 1570 *= start+$fc
EI 1580 stjump =*      ;these are the starting points
LE 1590 saddr  =*      ;specified in this file's header
NC 1600 lda #1
CJ 1610 sta erflag     ;for first call
NG 1620 jsr clrscr     ;give instructions
JN 1630 jsr doicon     ;do icons
GL 1640 rts            ; to main loop
FN 1650 ;--- initial screen ---
DH 1660 clrscr =*
BH 1670 lda #0
MJ 1680 jsr $c139      ; setpattern (setpat)
EO 1690 jsr $c19f      ; i.rectangle (pfill2)
GJ 1700 .byte 0,199
NO 1710 .wor  0,319
OO 1720 ; turn off the background screen
FA 1730 ; (we're using that ram)
GJ 1740 ; insert our own routine into the
AJ 1750 ; db screen recover vector.
AA 1760 lda disbuf
AN 1770 and #%10111111 ;bit 6=background enable
CF 1780 sta disbuf
HD 1790 lda #<recovr   ;our routine
GH 1800 sta recvec     ;geos vector
ED 1810 lda #>recovr
ME 1820 sta recvec+1
MJ 1830 ;
NG 1840 jsr $c1a2      ;i.framerectangle (pbox2)
LH 1850 .byte 0,16
KI 1860 .wor 144,319
FL 1870 .byte $ff      ;solid line
MJ 1880 ; print some user help info
KG 1890 ; first draw the 'fake' icons
IN 1900 jsr $c1ab      ;i.bitmapup (cbox2)
NG 1910 .wor iconch
PE 1920 .byte 3,60,6,16
CO 1930 jsr $c1ab
OL 1940 .wor larrow
LG 1950 .byte 3,84,3,16
AA 1960 jsr $c1ab
CO 1970 .wor rarrow
IE 1980 .byte 3,108,3,16
OB 1990 jsr $c1ab
AA 2000 .wor iconsa
NF 2010 .byte 3,132,6,16
HH 2020 ; tell what they do
PD 2030 jsr $c1ae      ; i.putstring (dsptx2)
DL 2040 .wor 24
OO 2050 .byte 40,24
FM 2060 .asc "icon definer  v1.0"
KA 2070 .byte 0
LH 2080 jsr $c1ae
FO 2090 .wor 24
PP 2100 .byte 52
OB 2110 .asc "by f. g. kostella for "
KG 2120 .byte 14       ; underline on
KC 2130 .asc "the transactor"
CH 2140 .byte 15,0     ; off
BM 2150 jsr $c1ae
JC 2160 .wor 80
LE 2170 .byte 72
PO 2180 .asc "a photo album"
CI 2190 .byte 0
DP 2200 jsr $c1ae
EG 2210 .wor 56
DJ 2220 .byte 96
PD 2230 .asc "previous album record"
EL 2240 .byte 0
FC 2250 jsr $c1ae
GJ 2260 .wor 56
NM 2270 .byte 120
EF 2280 .asc "next album record"
GO 2290 .byte 0
HF 2300 jsr $c1ae
PL 2310 .wor 80
LA 2320 .byte 144
AI 2330 .asc "current album record"
IB 2340 .byte 0
KB 2350 rts
LP 2360 ;--- icons ---
DE 2370 doicon =*      ;put up icons
KA 2380 lda #<myicon
OK 2390 sta r0
KB 2400 lda #>myicon
EI 2410 sta r0+1
GB 2420 jsr $c15a      ; doicons (cboxes)
KG 2430 rts
OP 2440 ;
HB 2450 myicon =*      ; icon tables
BD 2460 .byte 5,99,0,99 ; #, x&y pointer
KJ 2470 .wor iconch    ; graphic pointer
CG 2480 .byte 0,0,6,16 ; x,y,w,h dimensions
LG 2490 .wor choose    ; svc rtn pointer
DN 2500 .wor iconex
DL 2510 .byte 38,0,2,16
LP 2520 .wor doexit
MA 2530 .wor larrow
AB 2540 .byte 6,0,3,16
MN 2550 .wor dolast
AD 2560 .wor rarrow
HD 2570 .byte 9,0,3,16
OB 2580 .wor donext
OE 2590 .wor iconsa
PP 2600 .byte 12,0,6,16
FH 2610 .wor saveit
BF 2620 ;--- icon service routines ---
BL 2630 doexit =*      ; quit application
HC 2640 jmp $c22c      ; enterdesktop (restrt)
IC 2650 ;---
FI 2660 choose =*      ; choose a photo album
OA 2670 jsr findfl     ; put up file names
DD 2680 cmp #2         ; cancel selected
MM 2690 beq choos2
GP 2700 lda #0         ; start with rec #0
IA 2710 sta recnum
KI 2720 jmp getit
DE 2730 choos2 =*
AK 2740 rts
MI 2750 ;---
FJ 2760 donext =*      ; next record
GC 2770 inc recnum
HE 2780 bpl donex2
BN 2790 lda #0
CG 2800 sta recnum
LM 2810 donex2 =*
OO 2820 jmp getit
MN 2830 ;---
OD 2840 dolast =*      ; previous record
BF 2850 dec recnum
PP 2860 bpl getit
BC 2870 lda #0
CL 2880 sta recnum
AM 2890 ;
EP 2900 getit =*       ; put up filename
FN 2910 jsr doname
EH 2920 jsr getrec     ; and get the record
OF 2930 rts
KE 2940 ;---
DO 2950 saveit =*
OM 2960 jsr getfnm     ; prompt for save name
FF 2970 cmp #2         ; cancel selected
DD 2980 beq save2
KB 2990 jsr recbyt     ; translate
GD 3000 jsr savseq     ; save to disk
EP 3010 save2 =*
IL 3020 rts
MD 3030 ;--- setup & call db to get filename ---
LK 3040 getfnm =* ;user enters name of file
EE 3050 ;  clear out filename buffer
DO 3060 ldy #16
JO 3070 lda #0
```

```
LA  3080 getfn2 =*
EG  3090 sta svname,y
HN  3100 dey
LE  3110 bpl getfn2
PG  3120 ;  r5 to hold selected name in db
HE  3130 lda #<svname
GK  3140 sta r5
HF  3150 lda #>svname
MH  3160 sta r5+1
KI  3170 lda #<getndb    ; addr of db table
EM  3180 sta r0
GE  3190 lda #>getndb
KJ  3200 sta r0+1
JE  3210 jsr $c256       ; dodlgbox (window)
NN  3220 lda r0          ; .a holds #2 if cancel
FP  3230 rts             ; return it to caller
GH  3240 ;---
NA  3250 ; db table
JD  3260 getndb =*       ; return save name in r5 pointer
ON  3270 .byte $81       ; standard position
DE  3280 .byte 2,16,68   ; cancel icon
LB  3290 .byte 11,16,16  ; textstr cmnd
EN  3300 .wor gnmstr     ; pointer
JA  3310 .byte 13,16,30,$0c,16,0  ; getstr db cmnd ($0c = r5)
OG  3320 ;
DB  3330 gnmstr =*
GN  3340 .byte 24
BP  3350 .asc "please enter filename"
CI  3360 .byte 27,0
AK  3370 ;
EK  3380 ;--- setup & call db to find photo albums on disk ---
FA  3390 findfl =*
IG  3400 ;  clear out the phname
BE  3410 ldy #16
HE  3420 lda #0
NF  3430 fifl2 =*
DK  3440 sta phname,y
FD  3450 dey
AD  3460 bpl fifl2
LM  3470 ; search for this geos file type:
FF  3480 lda #7          ; appl. data
IA  3490 sta r7
GJ  3500 ; return selected file name in
CI  3510 lda #<phname
CC  3520 sta r5
CJ  3530 lda #>phname
IP  3540 sta r5+1
KE  3550 ; search for files with this
IA  3560 ; permanent name
BP  3570 lda #<permnm
GO  3580 sta r10
BA  3590 lda #>permnm
DN  3600 sta r10+1
AK  3610 lda #<ffildb    ; addr of db table
MH  3620 sta r0
KL  3630 lda #>ffildb
CF  3640 sta r0+1
HJ  3650 jsr $c256 ;dodlgbox
GG  3660 lda r0
NJ  3670 rts ; r0 is in .a
GN  3680 ;
FM  3690 ; db table
EA  3700 ffildb =*
KJ  3710 .byte $81       ; standard
CE  3720 .byte 16,4,4    ; getfiles cmnd
AE  3730 .byte 5,17,24   ; open icon
DO  3740 .byte 2,17,72,0 ; cancel
LM  3750 ;-----
EE  3760 ; print filename to screen
GJ  3770 doname =*
PM  3780 lda #9          ; horz lines
FO  3790 jsr $c139
EB  3800 jsr $c19f
BC  3810 .byte 0,15
BC  3820 .wor 145,303
BD  3830 jsr $c1ae       ; space, set pos.
OE  3840 .wor 160
JM  3850 .byte 10,32,0
LL  3860 lda #<phname    ;selected file
GH  3870 sta r0
AP  3880 lda #>phname
ME  3890 sta r0+1
GF  3900 jsr $c148
MA  3910 lda #47         ; slash
AL  3920 jsr $c145       ; putchar (dspchr)
FJ  3930 lda recnum      ; record #
ML  3940 sta r0
JF  3950 lda #0
CJ  3960 sta r0+1
GK  3970 lda #$c0        ; flush left
FG  3980 jsr $c184       ; putdecimal (dspnum)
AB  3990 lda #32 ;space
HL  4000 jsr $c145
GJ  4010 rts
CI  4020 ;---
NG  4030 dogrid =*       ; draw grid behind bitmap
HF  4040 lda #16
PH  4050 jsr $c139       ; setpattern
IG  4060 jsr $c19f       ; i-rectangle
CN  4070 .byte 16,199
IK  4080 .word 0,319
GO  4090 rts
IH  4100 ;--- error rtns ---
JH  4110 norec =*
AB  4120 jsr $c1ae       ; i.putstring
MD  4130 .word 110
FB  4140 .byte 102
KK  4150 .asc " empty record "
ED  4160 .byte 0
HD  4170 lda #1
EL  4180 sta erflag
HD  4190 jsr $c277       ; closerecordfile (vclose)
EF  4200 rts
FJ  4210 ;--
FN  4220 derror =*       ; err # is in .a
CA  4230 pha
MH  4240 jsr $c1ae       ; i-putstring
ON  4250 .wor 110
NI  4260 .byte 102
LP  4270 .asc " -disk error- #"
MK  4280 .byte 0
KE  4290 pla
IE  4300 pha
OC  4310 sta r0
LM  4320 lda #0
EA  4330 sta r0+1
IB  4340 lda #$c0        ; flush left
HN  4350 jsr $c184       ; putdecimal (dspnum)
PI  4360 lda #32
JC  4370 jsr $c145
JA  4380 lda #1
GI  4390 sta erflag
IF  4400 jsr $c277
EL  4410 pla             ; err 11 = too long
IK  4420 cmp #11         ; was the record too long
OA  4430 bne derr2
EE  4440 jsr $c1ae       ; i-putstring
MH  4450 .word 110
IF  4460 .byte 111
PF  4470 .asc " -record too long- "
EH  4480 .byte 0
FK  4490 derr2 =*
AI  4500 rts
EB  4510 ;
PM  4520 erflag .byte 0
ME  4530 ;--- draw the photo ---
KG  4540 drawph =*       ; draw selected photo
LA  4550 lda #>pstart
KO  4560 sta r0+1
AA  4570 lda #<pstart    ; skips w/h
MD  4580 sta r0
DL  4590 lda #0          ; x bytes pos
CF  4600 sta r1
JL  4610 lda #16         ; y pixel pos
IC  4620 sta r1+1
HF  4630 lda pload       ; width (at least 1)
```

```
NB  4640 beq badmap       ; not valid photo-might be the photo name strings
CP  4650 cmp #40
AO  4660 bcs badmap       ; too wide
KJ  4670 sta r2
FN  4680 lda pdepth       ; height lo-byte only
CA  4690 beq badmap       ; not valid
GA  4700 cmp #184
LJ  4710 bcs badmap       ; too long
OI  4720 sta r2+1
EO  4730 jsr $c142        ; bitmapup (cbox)
AH  4740 rts
IB  4750 badmap =*
EI  4760 jsr $c1ae        ; i-putstring
ML  4770 .word 110
FJ  4780 .byte 102
EG  4790 .asc " -bad bitmap- "
EL  4800 .byte 0
GL  4810 rts
IP  4820 ;----
CN  4830 ; when exiting a db, this rtn is called twice
KK  4840 ; to recover the db shadow & the db
HG  4850 recovr =*        ; called through $84b1
DD  4860 lda rvflag
EN  4870 bne norcvr       ; call once
NP  4880 lda #1
PI  4890 sta rvflag
JM  4900 jsr dogrid
AF  4910 lda erflag
AB  4920 bne nobmap       ; is there a bitmap
LK  4930 jsr drawph       ; then display it
AB  4940 nobmap =*
CE  4950 rts
BM  4960 norcvr =*        ; reset on 2nd call
FF  4970 lda #0
JO  4980 sta rvflag
KG  4990 rts
FN  5000 rvflag .byte 0
IA  5010 ;
DI  5020 ;--- disk routines ---
AP  5030 recnum .byte 0
EP  5040 ;opendrive =* ;optional
PJ  5050 ;lda #8 ; drive
MD  5060 ;jsr setdevice
FM  5070 getrec =*
NH  5080 jsr dogrid
ME  5090 jsr $c2a1        ; opendisk (opndsk)
CE  5100 lda #<phname     ; name of album
OE  5110 sta r0
IM  5120 lda #>phname
EC  5130 sta r0+1
EO  5140 jsr $c274        ; openrecordfile (vopen)
BM  5150 txa              ; x=0 if no error
HD  5160 beq grc1
DM  5170 jsr derror
IC  5180 rts
PG  5190 grc1 =*
EI  5200 lda recnum
CF  5210 jsr $c280        ; pointrecord (goto)
LB  5220 tya
JA  5230 bne grc2         ; 0 if empty
GN  5240 jsr norec
AK  5250 rts              ; rec empty
GL  5260 grc2 =*
NL  5270 ; ok, now read it in
EK  5280 lda #$15         ; max # of bytes (=sbegin-pload)
IM  5290 sta r2+1
DB  5300 lda #$d0
KB  5310 sta r2
BB  5320 lda #>pload      ; load to address
KP  5330 sta r7+1
JA  5340 lda #<pload
ME  5350 sta r7
HJ  5360 jsr $c28c        ; readrecord (vload)
GP  5370 ; .x hold error #
IL  5380 txa
FC  5390 beq grc3
JK  5400 jsr derror
OA  5410 rts
```

```
HF  5420 grc3 =*
DK  5430 ;r7=addr of first byte following the last byte read in
AB  5440 lda r7           ; we'll use a4 as a pointer
BL  5450 sta a4           ; (r7 is destroyed, a4 is reserved for our use)
OD  5460 lda r7+1
PG  5470 sta a4+1
DF  5480 lda #0
OJ  5490 sta erflag       ; if it got this far!
ID  5500 jsr drawph
PF  5510 jsr $c277        ; closerecordfile (vclose)
MH  5520 rts
JM  5530 ;----------------
CC  5540 pcount .byte 0
BJ  5550 recbyt =*
NC  5560 ; photo loaded at pload to the addr in a4 -1
LN  5570 ; the bytes file will be saved from sbegin to a3
MG  5580 ; set up a2 (photo pointer) & a3 (bytes pointer)
DA  5590 lda #<pload
LC  5600 sta a2
DB  5610 lda #>pload
BA  5620 sta a2+1
GN  5630 lda #<sbegin
FF  5640 sta a3
GO  5650 lda #>sbegin
LC  5660 sta a3+1
BH  5670 ldy #0
DE  5680 sty pcount
MN  5690 xbyt1 =* ;loop
JL  5700 lda pcount
BG  5710 and #$0f
JO  5720 bne xbyt2        ; every 16 bytes, start a new line
LJ  5730 lda #13          ; eol
OM  5740 jsr addchr
CM  5750 lda #46          ; period
CO  5760 jsr addchr
BF  5770 lda #66          ; b
GP  5780 jsr addchr
JI  5790 lda #89          ; y
KA  5800 jsr addchr
JI  5810 lda #84          ; t
OB  5820 jsr addchr
FP  5830 lda #32          ; spc
CD  5840 jsr addchr
PP  5850 jmp xbyt3
KF  5860 ;
PO  5870 xbyt2 =*         ; use a comma
HI  5880 lda #44
EG  5890 jsr addchr
GG  5900 xbyt3 =*
LK  5910 inc pcount
KO  5920 lda (a2),y       ; index into photo file
EA  5930 jsr bythex
NO  5940 jsr inca2
NM  5950 jsr cmpa24       ; done yet
JA  5960 bcc xbyt1
IN  5970 lda #13
OL  5980 jsr addchr
MO  5990 lda #13
LN  6000 sta (a3),y
IH  6010 lda #<sbegin     ; save start in header
NB  6020 sta sstart
CG  6030 lda #>sbegin
DP  6040 sta sstart+1
HM  6050 lda a3           ; save end in header
CC  6060 sta seqend
HI  6070 lda a3+1
CE  6080 sta seqend+1
GL  6090 rts
KE  6100 ;
GD  6110 ; translate a byte into hex format
AL  6120 bythex pha       ; save byte
FI  6130 lda #36
OF  6140 jsr addchr
OI  6150 pla
MI  6160 pha
HF  6170 lsr              ; move hi-nybble into low
NP  6180 lsr
HA  6190 lsr
BB  6200 lsr
```

```
LK  6210 jsr fndasc      ; returns with ascii char in .a
NK  6220 jsr addchr      ; write it to buffer
GM  6230 pla             ; get original byte
EL  6240 jsr fndasc
OL  6250 jsr addchr      ; write it
AG  6260 rts
CK  6270 ;----
HF  6280 fndasc =*       ;  returns ascii for 4 lsb in .a
EN  6290 and #$0f        ; clear 4 hibits
EP  6300 cmp #$0a        ; >9 print
ID  6310 bmi find1
NC  6320 adc #$06        ; it's a-f ; +$36
AH  6330 find1 =*        ; if it's 0-9, add $30 to convert to ascii
CO  6340 adc #$30
KL  6350 rts
OE  6360 ;
IL  6370 addchr =*       ; add char in .a to the byte (file) buffer
HF  6380 sta (a3),y
AL  6390 jsr inca3
MO  6400 rts
AI  6410 ;
PK  6420 inca2 inc a2
FB  6430 bne xia2
HB  6440 inc a2+1
PN  6450 xia2 rts
CL  6460 ;
GO  6470 inca3 inc a3
LE  6480 bne xia3
LE  6490 inc a3+1
CB  6500 xia3 rts
EO  6510 ;
KN  6520 cmpa24 =*
OO  6530 ; if a2 >or= a4 then we've exceeded the eof so set .carry
EB  6540 ; if < then return with .carry clear
MA  6550 ;
BC  6560 sec
HL  6570 lda a2
FN  6580 sbc a4
JG  6590 sta xxtemp
HJ  6600 lda a2+1
FL  6610 sbc a4+1
LC  6620 ora xxtemp      ; flags set
CN  6630 rts
GG  6640 ;
NL  6650 xxtemp .byte 0
KH  6660 ;
KI  6670 ;-------
NB  6680 savseq =*
FI  6690 lda #<header    ; header block for file
GJ  6700 sta r9
PO  6710 lda #>header
MG  6720 sta r9+1
FD  6730 lda #0
OD  6740 sta r10
IE  6750 jsr $c1ed       ; savefile (save)
EF  6760 rts
IO  6770 ;
IM  6780 ;--- icon graphics ---
EM  6790 iconex =*       ; exit icon
NK  6800 .byte 160,255,255,0,0,255,255,128,1,128,1,128,1
AA  6810 .byte 143,241,143,241,143,241,128,1,128
FN  6820 .byte 1,128,1,255,255,0,0,255,255,0,0
DE  6830 ; the icons below were "grabbed"
OG  6840 ; with an earlier version of this
AP  6850 ; program.
GC  6860 iconch =*       ; choose icon
KM  6870 .byte$05,$ff,$82,$fe,$80,$04,$00,$82
IG  6880 .byte $03,$80,$04,$00,$b8,$03,$8f,$98
NB  6890 .byte $00,$00,$00,$03,$98,$d8,$00,$00
AK  6900 .byte $00,$03,$98,$1f,$1e,$3c,$78,$f3
GC  6910 .byte $98,$1d,$b3,$66,$cd,$9b,$98,$19
CE  6920 .byte $b3,$66,$c1,$9b,$98,$19,$b3,$66
GC  6930 .byte $79,$fb,$98,$19,$b3,$66,$0d,$83
PK  6940 .byte $98,$d9,$b3,$66,$cd,$9b,$8f,$99
PO  6950 .byte $9e,$3c,$78,$f3,$80,$04,$00,$82
LJ  6960 .byte $03,$80,$04,$00,$81,$03,$06,$ff
MN  6970 .byte $81,$7f,$05,$ff
KL  6980 ;
DG  6990 iconsa =*       ; save icon
ME  7000 .byte $05,$ff,$82,$fe,$80,$04,$00,$82
CM  7010 .byte $03,$80,$04,$00,$b8,$03,$80,$1f
IG  7020 .byte $00,$00,$00,$03,$80,$31,$80,$00
PK  7030 .byte $00,$03,$80,$30,$1e,$66,$78,$03
BC  7040 .byte $80,$30,$33,$66,$cc,$03,$80,$1f
LF  7050 .byte $1f,$66,$cc,$03,$80,$01,$b3,$3c
II  7060 .byte $fc,$03,$80,$01,$b3,$3c,$c0,$03
KG  7070 .byte $80,$31,$b3,$18,$cc,$03,$80,$1f
HO  7080 .byte $1f,$18,$78,$03,$80,$04,$00,$82
NB  7090 .byte $03,$80,$04,$00,$81,$03,$06,$ff
OF  7100 .byte $81,$7f,$05,$ff
MD  7110 ;
BM  7120 larrow =*       ; left arrow icon
LF  7130 .byte $b0,$ff,$ff,$ff,$80,$00,$03,$80
JO  7140 .byte $00,$03,$80,$00,$03,$80,$40,$03
CK  7150 .byte $81,$c0,$03,$87,$ff,$f3,$9f,$ff
PH  7160 .byte $f3,$87,$ff,$f3,$81,$c0,$03,$80
HA  7170 .byte $40,$03,$80,$00,$03,$80,$00,$03
HO  7180 .byte $80,$00,$03,$ff,$ff,$ff,$7f,$ff,$ff
MI  7190 ;
AG  7200 rarrow =*       ; right arrow icon
KK  7210 .byte $b0,$ff,$ff,$fe,$80,$00,$03,$80
NC  7220 .byte $00,$03,$80,$00,$03,$80,$04,$03
DL  7230 .byte $80,$07,$03,$9f,$ff,$c3,$9f,$ff
AJ  7240 .byte $f3,$9f,$ff,$c3,$80,$07,$03,$80
LE  7250 .byte $04,$03,$80,$00,$03,$80,$00,$03
HD  7260 .byte $80,$00,$03,$ff,$ff,$ff,$7f,$ff,$ff
MN  7270 ;
MF  7280 permnm =*       ; find any version
IK  7290 ;.asc "photo album "
DH  7300 .byte $70,$68,$6f,$74,$6f,$20,$61,$6c,$62,$75,$6d,$20
EF  7310 .byte 0 ;(version not included)
HB  7320 ;  the new photo album (v2.1) stores the
NC  7330 ;  names of the individual photos as the
OE  7340 ;  last used record.
NF  7350 ;  use: .asc "photo album v1.0"
LE  7360 ;  to read the older versions exclusively.
FH  7370 ;  use: .asc "photo album v2.1"
BN  7380 ;  to read the newer version.
EF  7390 ;
KB  7400 phname .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; selected photo album
NF  7410 svname .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; user entered name
ON  7420 ;--------------------------
GD  7430 ; this header is saved along with the file
LK  7440 header =*
PP  7450 .wor svname
MJ  7460 .byte 3,21 ,$bf ; 3x21 icon
OK  7470 .byte $ff,$ff,$ff,$80,0,1,$80,0,1,$80
BI  7480 .byte 0,1,$80,0,1,$80,0,1,$80,0,1,$80,0,1
GJ  7490 .byte $80,0,1,$80,0,1,$80,0,1,$80,0,1,$80
FJ  7500 .byte 0,1,$80,0,1,$80,0,1,$80,0,1,$80,0,1
PJ  7510 .byte $80,0,1,$80,0,1,$80,0,1,$ff,$ff,$ff
AG  7520 ;this icon is just a square outline
JK  7530 ; save the file as a c= seq
MC  7540 .byte $81        ;  c= seq
HF  7550 ; actually, when the kernal writes
NJ  7560 ; the file to disk, it wont save
LB  7570 ; the header to disk
GP  7580 ; when it sees the next
AI  7590 ; byte, a filetype of non-geos.
AN  7600 .byte 0          ; non-geos/alternately use 3 for data
EH  7610 .byte 0          ; geos seq
IK  7620 ; next two words placed by translate rtn
CB  7630 sstart =*
PI  7640 .wor 0           ; start addr
DI  7650 seqend =*
DH  7660 .wor 0,0         ; end addr, jump addr
PK  7670 .asc "icon definerv1.0"
MJ  7680 .byte 0,0,0,0
AI  7690 ;
CJ  7700 .asc "f. g. kostella  "
KL  7710 .byte 0,0,0,0
OJ  7720 ;
JN  7730 *=*+139
EF  7740 endcod =*        ; specified in header
CC  7750 .end
```

RAMifications

Approaches to fattening the C128, with critique and comments from Paul Bosacki, who did it for the 64

by Richard Curcio

Paul Bosacki's C256 (Volume 9, Issue 2) is the most exciting hardware project for the C64 to come along in a long time. I offer a bit of advice to others who would modify a valuable piece of equipment:

Don't plan on re-using chips desoldered from equipment. Often, the efforts to remove the chip intact result in damage to the circuit board. If the IC is successfully removed, one can never be 100% certain of its reliability. Without access to professional desoldering equipment, it's better to plan on discarding any removed chips. Having made this decision, chip removal becomes much easier.

On the solder side of the circuit board, some pins may be bent over. Straighten these by heating with a low-wattage soldering iron and slipping the blade of a small jeweller's screwdriver or a hobby knife (X-acto) between the bent-over portion and the PC board. (This step is essential if you do decide to risk re-using the chips.) On the component side of the board, using a sharp, small diagonal cutter, clip each IC pin as close as you can get to the body of the chip.

Once all the pins of one chip have been clipped, and the body discarded, begin desoldering the pins. Heating each pin from the bottom of the board, most pins will simply drop out of the hole. Those that don't can be removed from the top-side. Grab each pin with small needle-nose pliers or tweezers while heating the pin with the soldering iron. With just a few jiggles, the pin should come free.

Clear out the hole from the solder side with a "solder sucker". Radio Shack carries a very inexpensive one. The trick with these is to keep the nozzle clear. Empty the sucker often and use a tooth-pick or straightened paper clip to clear the nozzle. A sharpened wooden tooth-pick is also useful for clearing out the PC board holes.

Double-sided printed circuit boards (circuit traces on both sides) use "plated-through" holes. In other words, the inside of the hole is copper clad to continue the circuit from one side of the board to the other. If any desoldered component lead or IC pin comes out with a small copper-coloured cylinder on it, the through-plating has come loose. Usually, soldering the replacement part on both sides of the board will restore the circuit's continuity. If you can't get to the component side of the part (such as the underside of an IC socket,) flow just a *little more* solder than necessary from the bottom of the board.

You might want to practise these techniques before mangling your trusty C64. Most large cities have an area that used to be known as "Radio Row", where one can purchase everything from ancient teletype machines and radar transmitters to obsolete and/or defective consumer electronics. Pick up a few junk circuit boards and try your hand at removing ICs.

RAMifications: the C128

There are a number of possibilities (with attendant difficulties) in performing a memory expansion similiar to Paul Bosacki's on the C128. The following is the result of many nights spent studying the C128 schematics and is intended to stimulate some debate on the subject.

As many functions as is reasonable should be controlled by software, with a few key functions controlled by switches that over-ride the software settings. There is ample I/O space for adding latches and such without using the 'official' I/O locations. At least part of the added memory should be available in 64 mode and be compatible with Mr. Bosacki's design.

With these criteria in mind, consider these suggestions:

1) Replace the 64K of RAM 1 with 256K chips, dividing RAM 1 into 'sub-banks'.

In 128 mode, we need not be concerned with the location and amount of common RAM since it's always in RAM 0 and the MMU will take care of everything. Pin 47 of the MMU, variously referred to as MS3 or c128, goes to logic 0 in 64 mode, disabling the fast serial circuitry and signalling the PLA to assume its C64 persona. This can be used to disregard RAM 0 and switch over to the expanded RAM 1 in 64 mode and enable the CRAM circuitry that would be needed, since the MMU effectively vanishes.

However, we would lose the ability to have something in RAM 0 be present when we switch to 64 mode. If we attempt to make RAM 0 available, along with the expanded RAM 1, in 64 mode we then have five 64K banks - a major incompatibility with Mr. Bosacki's design.

2) Replace both RAM 0 and 1 with 256K chips, creating a C512!

Besides the potential for damaging the circuit board by desoldering 16 sixteen pin chips (that's 256 pins, folks), we have to duplicate the CRAM actions of the MMU. There is no signal from the MMU to indicate that common RAM is being accessed. The task is to capture the common RAM amount and location(s) values written to the RAM configuration register at $D506 (not overly difficult; $D5xx is already decoded), and qualify our CRAM enable with the CAS intended for RAM 0. Appropriate three-state circuitry, controlled by MS3, could allow this mod to mimic Mr. Bosacki's modification in 64 mode.

3) Add two 64K banks, comprised of four 64Kx4 DRAMs.

In addition to capturing the CRAM parameters, if we also capture bit 7 of $FF00, we can implement the missing banks 2 and 3, which the operating system already knows about but the MMU and the rest of the hardware don't provide. Only minor alterations need be performed on the C128 PC board. More RAM could be added at a later date. The hardest part may be getting it all to fit inside the case.

4) Don't add any RAM. Instead, make RAM 1, the second colour memory bank, and possibly the fast serial circuits, available in 64 mode.

This is pretty simple. Commodore could have made this available as an 'enhanced' 64 mode.) When compared to a C256, this is pretty meagre fare, and may not be worth the effort. On the other hand, the exercise may be useful as a preliminary hack.

5) Do nothing. Wait for Commodore to produce the C1000. (Don't hold your breath.)

Personally, I am inclined to go with item one as a plan of action. Memory is increased by 196K to a total of 320K. Dealing with the difficulties of item two may require an inordinate amount of logic. Initially item three merely doubles the amount of memory and requires even more 'outboard' electronics.

There are numerous details to be worked out in implementing any of the above (except for item 5): What is the difference between MUX, which doesn't exist in the C64's old VIC chip, and CAS? Should the new CRAM circuitry look at the processor address bus, or the translated address bus? How should the preconfiguration registers be handled, if at all? How should expanded memory behave in Z80 mode?

With enough hardware, anything is possible. The question is: Is it practical?

Dialogue

The following text is taken from an exchange of letters between Mr. Curcio and Mr. Bosacki. Mr. Curcio, like Mr. Bosacki before him, has kindly agreed to the publication of his address. - MO

I applaud your accomplishment, creating a fat C64. I am eager to know the details of the 512K expansion. If it uses a switch to select the second 256K, you might be interested in Figure 2 of the enclosed schematics. I am told that your modification doesn't work on the newer 64Cs. Can you confirm this or provide a fix? *[Happily, Mr. Bosacki has obtained an E-board 64C and states that the mod can be achieved in that machine also. -Ed.]* Also, how are RAM-based characters handled by your circuit? Does *Configure 256* work with *geoPaint*? How can we implement your GEOS mod without the (alleged) benefit of *GeoProgrammer*? I am not a fan of GEOS and if you can provide any improvements to it (like turning off proportional spacing in text in *geoPaint*) that would be great.

Have you considered the possibility of a C128 expansion? I have a few ideas regarding that, and it seems that there are more than a few difficulties to overcome.

I had heard about your C256 some months before seeing it in *Transactor*. While awaiting that issue, I devised (on paper) my own method for RAM expansion. I don't know if it would work, and I admit that my knowledge of dynamic RAM is somewhat

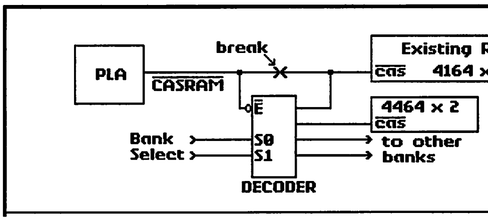
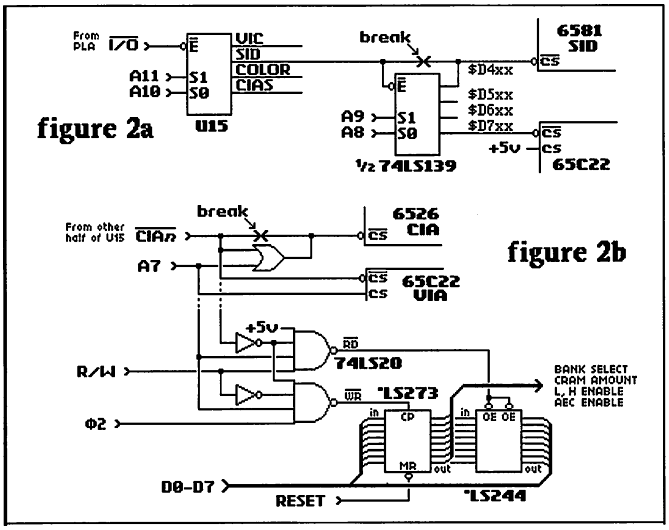


figure 1



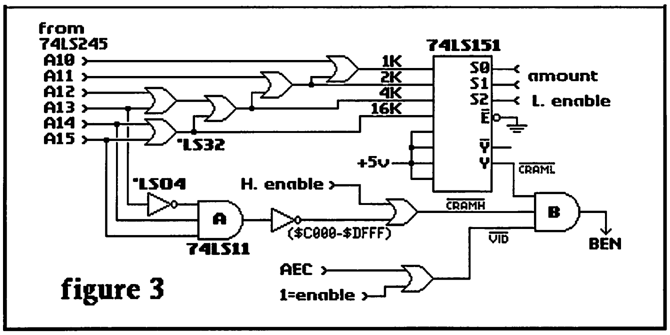
sketchy. My method requires a bit more hardware and wiring than yours. It would fit inside a C64 but probably not a 64C.

Basically, my scheme (Figure 1), would leave the original RAM chips in place, and divert the PLA's CASRAM signal from the 4164s to the enable of a 74LS139 or similar decoder. One decoded output goes to the on-board memory chips and the others go to banks of 4464s (64K x 4). The value on the decoder's select bits determines which bank gets CAS'ed. (A three-bit decoder would provide 8 64K 'banks'.) Any Common RAM or Bank Enable signal would force this value to select the original RAM. Memory could be expanded in stages, and the risk of damage to the circuit board is minimized by not desoldering any chips.

On the other hand, all the data and MA lines would have to be brought to the expander board. Looking at your design, I get the uneasy feeling that mine wouldn't work. Am I mistaken in assuming that unselected banks would continue to be refreshed via RAS?

A master 'mod disable' switch is a good idea. I've gotten used to setting switches to configure my VIC-20, but for the C64 switches seem like several steps backwards.

The features I have in mind need more bits than those provided by the cassette lines of the 6510 port. To create more I/O space to supply the needed bits, Figure 2a decodes SID's enable (which occupies a wasteful 1K of I/O), into four pages. One of these is used to enable a 65C22 or other parallel interface chip.



This would power up or reset with its port pins high (via pull-up resistors) and the logic would be such that this selects the normal memory configuration. Software would be compatible as long as it doesn't use any SID 'image' addresses. The 65C22 is undoubtedly over-kill for this application (the CMOS version, to reduce current consumption, is also hard to find), but it's easily interfaced and its other port might come in handy, perhaps for selecting alternate ROMs.

The alternative is to also decode R/W and phase 2 to provide a WRite strobe for a latch and a ReaD for a tri-state buffer to read back the latch contents. The latch should have a "clear" input so that RESET provides all zeroes as the default.

The 65C22 would need to be initialized. On reset, the ports go high because the DDRs (and all other registers) are cleared, putting the ports in input mode. Changing the DDRs to output 'cold' would cause the port pins to go low because the Output Registers were also cleared. Fortunately, the 65C22 permits writing to the Output Registers while configured for input. Why isn't this a problem with the MPU port? (Or is it?)

Another way to obtain more I/O space is shown in Figure 2b. Here one of the CIA enables is intercepted and decoded with A7 to provide a new enable. I tend to favour this approach, and the lower half of the figure shows the latch/buffer alternative to the 65C22. If Bank Select remains at the 6510 port, three of the latch bits are unused and available for other tricks.

I would like to have different amounts of common RAM, perhaps as much as 16K so that a bit-map at $2000 could be seen from the other banks. In Figure 3 the OR gates of your design are slightly rearranged to provide a logic 0 when the lowest 1, 2, 4, or 16K of memory are addressed. Bits S0 and S1 of the 74LS151 data selector choose how much *low* common RAM will be present. Bit S2 can force the '151 to look at the pins held at +5V, causing CRAML to be high. (The E input can't be used because when the '151 is disabled, output Y is low - which is the opposite of what's needed, although it could be used for a software 'mod disable'.) With no low common RAM, each bank could thus have its own zero-page and stack!

Because of the scarcity of free RAM in the lowest 1K of memory, Figure 3 also provides the capability of common RAM in high memory. Three-input AND gate A, with two left-over inverters, supplies CRAMH when A15, 14, and 13 are 110. This decodes addresses $C000-$DFFF as common RAM when H enable is low. Of course, D-block is accessible only when I/O and character ROM are switched out. Gate B replaces the two-input AND gate in your circuit. The remaining OR gate can be used for a software AEC enable. (In your circuit, how does an open switch translate to AEC enabled, since AEC doesn't matter?) Add inverters as necessary to provide the power-up configuration - depending upon a 65C22 port with all port bits at 1 or a cleared latch with all zeroes. That leaves one idle three-input and two 2-input ANDs for any other enhancements. In this circuit the amount of High Common RAM is fixed, but added complexity could provide variable amounts.

When things become too complex for gates and decoders, some means must be found to reduce the chip count. I include a data sheet for a UV erasable PLA. This device was sold by Jameco a while ago. I don't know if it's still available. Supposedly, this device could be programmed in an EPROM burner(?). If it doesn't come out right, erase it and try again. I assume the manufacturer can provide more information regarding programming, etc.

Even without resorting to a home-burned PLA, I'm sure that some of the above could be done more efficiently. Every time I thought that this letter was finished, a new idea would pop into my head. Thank you for a most stimulating project. I can hardly wait to try this stuff.

Richard Curcio
22 Seventh Avenue
Brooklyn, N.Y. 11217

Thank you for your response to my article. The big kick with this article has not been getting it published, but rather receiving letters like yours. Letters from people who understand the workings of the project. My wife, I'm afraid, has gotten a bit tired of hearing about AND gates and *CRAM and "two bit codes" etc.

But, on to your questions. The most important of which is: Is the project compatible with the newer 64s. The preliminary answer is a cautious "yes". The newer models use LH2464's (read 41464), a 64x4K bit DRAM. The modification, as I see it at this point, is simple. The 'LS157 multiplexor is replaced with an 'LS139 which is permanently enabled. The two pseudo addresses are then used to drive the select pins of one half of the 'LS139. The four output pins are then connected to four banks of 2x64x4K bit DRAMs. The problem here is the additional wiring of the DRAMs. All this is, as I said, preliminary; I haven't tried it yet, and I'm still in the process of drawing up the schematics.

The other possibility here is forsaking the two x 4 DRAMs on the board and laying in a bank of 41256's. Although the layout of the system board is quite different than that of the old 64s, all the signals needed are directly accessible. My opinion is that this is the way to go. But again, I haven't tried it yet. And DRAM is expensive right now (I got my first 256K for $50).

Your next question is a bit unclear. Does this RamEx work with *geoPaint*? I take it to mean, do you get the fast DMA that allows the quick scrolling about a *geoPaint* document? If that's your question, the answer is no. As I mentioned in the article, the stash, fetch, swap and verify routines are not supported. Still vague? The DMA routines in GEOS are based on the principle that it's faster to use the REU to move data than it is to use the MPU. And they're right. My RamEx, however, uses the MPU to move data, and since it moves it across banks, that move is a little slower than a move within one bank. So, for that reason, I chose not to support those routines. For that reason and another: you can only modify so much GEOS code before you run out of space.

To implement the GEOS mod without *geoProgrammer:* Use *PAL* or some other assembler (I used to use *Buddy*). Type in code remembering to expand all macros, shorten the routine names, and make all other changes appropriate to your assembler. Then assemble and split the the header manually from the program. Take a look at the GEOS program from the previous *Transactor. [This is a reference to the "maketogeos" program and PAL-format source code that was featured in "Programming in GEOS" by Francis G. Kostella in **T** 9:1. - MO]* That's how they did it.

Concerning the 'alleged' advantages of *geoProgrammer*. Having used it, *Buddy* and *PAL*, I wouldn't go back. It's just too good. And, of course, for programming under GEOS, just too necessary!

I know it's impossible to get text to line up properly in *geoPaint*. If you want to get rid of proportional spacing, try using the Commodore font. It's an ugly font, but it will do just that. If your problem is getting text to line up, use the edit tool and just move it into place. Once you develop the knack, it's quite easy to do.

About expanding the 128: my only thought there used to be this: piggyback the DRAMs with 4164's, bend up pin 15 and connect it to the MMU's *CAS2, and *CAS3 output. That's an easy 256K, with 128K unused in any way by the OS. Other than that, I don't have access to schematics for that machine and am unfamiliar with its architecture. Will my above suggestion even work? Isn't that part of what you're getting at in your comments? You're better qualified here than me. Although I would love to get my hands on the schematics for that machine. Got an order number?

Something I want to get at here about your modification. It seems that it would work, yes. And, yes, the RAM would be refreshed via *RAS. But don't overlook the C64 power supply. One reason I went for replacing the DRAMs altogether was that power supply. Too much support circuitry and two much DRAM equals one dead power supply, and that dead power supply will propably have taken the DRAM with it. Something to consider! My board does not require a beefier power supply (except in those marginal cases). My 64 ran for six months before I got an REU, with no problems. And then I went to 512K, which definitely requires the heavier power supply.

Although it's certainly not obvious, I had two overriding concerns when I went at this expansion project. Keep the parts count down and keep it transparent. Using the MPU port literally required the master disable switch. And the other switches? I agree, switches are a step in the wrong direction. But I wanted to show that it could be done simply. The other option, even when that board was being developed, was an 8-bit latch mapped into phantom I/O space, but the additional circuitry was too much a complicating factor. I thought that many more people would try to build a six-chip board than a twelve. Yes, my first 512K had 12 chips and was a patch on the original board.

Paul Bosacki

How Random is RND?

An analysis of the C64 and C128 RND routines

by D. J. Morriss

The beginner in BASIC is often surprised to find the RND function. What is the use of a function that supplies an allegedly completely unpredictable number? The more experienced programmer sees the utility of such a function in games; or, more seriously, in simulations. But some new questions occur: how does the computer generate these random numbers, and are they truly random, or do they exhibit some hidden bias? You're about to implement a cost-cutting inventory control system, based on your simulation of typical demand levels; will it cut costs or lose sales? Or you're specifying maximum drainage capacity for a proposed municipal storm-sewer system, after simulating 200 years of rainfall extremes on your computer. This could get expensive!

Part of the problem is that a random-number generator is, by its very nature, difficult to test. If you come up with a great new square-root routine, you need try only a few values to know if it works properly. But a random number generator can be shown to be 'good' only by examining many obscure properties of long sequences of numbers produced by the generator. Most of us lack both the skill and time to rigorously test a RND function, preferring to trust in the designers of the BASIC Interpreter. Unfortunately, the RND function, as implemented on the C64 and C128, has a number of serious shortcomings that should be recognized if RND is to be used effectively.

In theory, a computer is a completely determined system: given the initial state, and in the absence of such variables as keyboard input, the subsequent states of the computer are completely defined. Such a system cannot generate true random numbers; instead, RND produces pseudo-random numbers. They pass (or should pass) all the tests of random numbers, such as average value, standard deviation, and runs up and down, but they are generated by some specific mathematical technique. The proof that these are not true random numbers lies in the fact that the same sequences can be created over and over again. In fact, there are three separate mathematical techniques used to generate pseudo-random numbers on the C64 and C128, sometimes in conjunction.

The first is juggling: the digits that make up the number are rearranged in some defined pattern. In the C64 and C128, this juggling is done on the basis of individual bytes of a four byte number. The second system is multiplication and addition: some starting number is multiplied by one constant, and another constant is added. The resulting random number is perhaps altered in some other way, and becomes the starting point for a repeat of the same process. The third system involves passing the buck; the computer tries to find some external source of random numbers that it can use to generate its internal random numbers.

The RND function on the C128 is located in ROM, starting at $8434, in Bank 15, and is the same in both Versions 0 and 1 of the ROMs. The routine makes extensive use of Floating Point Accumulator #1, (FPA #1), where BASIC stores intermediate results, located from $63 to $68. Another important storage area is called RNDX, located from $121B to $121F. This is the mysterious random number seed, from which new random numbers grow. Listing 1 is a commented disassembly of the C128 routine. Subroutines used by RND are not included, but their purposes are indicated.

The C64 RND routine is located at $E097, FPA #1 is at $61 to $66, and RNDX is at $008B. Although some of the details and subroutine calls are different, the general flow of the C64 RND routine is the same as the C128 version. Listing 2 is a commented disassembly of part of the C64 RND function. Most of the following discussion of the C128 routine applies equally to both computers; where differences are significant, they are noted.

The C128 routine starts at $8434, by determining the sign of the argument of RND, stored in FPA #1. The routine then takes three different routes, depending on whether that argument is: a) negative; b) zero; or c) positive.

RND(-)

(The C128 and C64 versions of this part of RND are identical.)

If the argument of RND is negative, the C128 routine branches to $846A. Here, the first and fourth bytes of FPA #1 are interchanged, then the second and third. At this point, FPA #1 contained the negative argument of RND. The sign byte is set to zero, making the value positive, and the exponent byte is transferred to the rounding byte. The exponent byte is then set to $80, meaning a number between one-half and one, and the number is "normalized". The process of switching bytes may have led to a number in which the most significant bit (or bits) of the most significant byte is not a one. During normalization, the whole four-byte number in FPA #1 is shifted left, and the exponent decreased by one for each shift, until that most significant bit *is* one. The new bits needed at the least significant end of the number (if any) are obtained from the rounding byte. This new number in FPA #1 is left there as the value returned by the RND function, but a copy is packed (stored in a slightly different way) in RNDX, starting at address $121B, for reasons that will (I hope) become clear soon.

Thus, we see that RND of a negative number creates a random number by carrying out a specific manipulation on that negative number. So, RND(-3) will always yield the same number, and RND(-112233) will always yield some other specific number. RND(-) on the C64 follows exactly the same procedure: a given negative argument gives the same random number on both machines.

As the disassembly shows, the least significant byte of the argument becomes the most significant byte of the resulting random number. If the argument is a number with only a few digits (when expressed in binary floating point form), then the least significant byte (or bytes) will be zero. Such arguments include integers, and fractions like 0.1328125 (17/128) and 0.890625 (57/64). When these zero bytes are switched to become the most significant bytes of the random number, a lot of normalization is necessary. This in turn means that the exponent is decreased a lot, creating a very small random number. RND(-37), for example, is 3.45808076E-8, and RND(-6.625) is 4.94792403E-8. On the other hand, the binary representation of 37/131, for example, uses all four bytes, and RND(-37/131) is a more reasonable 0.985681329. (Can any random number be termed reasonable?)

How does RND(-37/131) compare to RND(-37/131/2) or RND(-37/131*64)? They are the same! The only difference in the binary floating point form of the three arguments is in the exponent, and the exponent gets shifted to the rounding byte, to be used as the source for new bits during normalization. Since these arguments use all four bytes, little normalizing is required, and the rounding byte is probably not needed. On the other hand, RND(-5), RND(-10), and RND(-20), although all very close, are different. The different exponents of the arguments, stored in the rounding byte, become significant when the random numbers, with only a few bits set, are normalized.

I have often seen articles which recommend RND(-TI) as a good source of really random numbers. Since the jiffy clock keeps time in integral numbers of jiffies, it is clear that RND(-TI) will always yield very small random numbers. Perhaps a variation like RND(-TI*SQR(2)) would give the desired result.

Since RND(-) always gives the same value for a given negative argument, it would not appear to be a very useful random-number generator. Indeed, about the only purpose of RND(-) is to store a new, definite value in the RNDX area.

RND(0)

(The C128 and C64 versions of this part of RND are significantly different.)

Referring to the disassembly of the RND routine again, if the argument of RND is zero, the C128 routine falls through to $843B. Here, the routine selects Bank 15, then proceeds to load FPA #1 with the contents of $DC04 to $DC07, and jumps to $847A. This is the same part of the routine that sets the sign positive, the exponent byte to $80, normalizes the value, copies it to $121B, and then exits.

The locations that produce the random number, $DC04 to $DC07, are four registers on Complex Interface Adapter (CIA) #1. These four bytes store the high and low bytes of two internal 16-bit timers. They are not part of the Time Of Day Clocks, as some references have stated (but see the C64 version discussion below). The timers are normally used to count pulses of the system clock. In theory, each byte could have any value from 0 to 255, giving totally random numbers; however, only one of these two timers is running! The other timer, at $DC04-05, is used in the C128 for tape and fast disk operations, and is left stopped by these routines, usually reading 2 or 1. Enter and run this short program to display the four timer bytes.

```
100 bank 15 : sa = 56320
110 for k=4 to 7 : print peek(sa + k),
120 next : print
130 go to 110
```

As you can see, only two of the four bytes are changing. This limits the range of possible values that will be transferred to FPA #1, and thus limits the random numbers produced by RND(0). Since only the first and third bytes of the random number are changing, while the second byte is a constant, and the fourth is zero, the result is a set of random numbers clustered closely (but not very, very closely) about 256 evenly spaced values.

To demonstrate the effect of this stopped timer, run this short program:

```
200 print rnd(0) * 256 : go to 200
```

Note that all the numbers printed are very close to whole numbers. In fact, if the stopped timer is reading 1, the fractional part of the number will be between 0.00390625 and 0.0078125; that is, between 1/256 and 2/256.

This limitation seriously decreases the usefulness of RND(0). For example, you may wish to generate a random integer between 1 and 1000 inclusive; a statement like:

```
x = int (rnd (0) * 1000 + 1)
```

would appear to do exactly this. In fact, only 256 of the 1000 integers would ever be generated!

This is a very good reason to avoid RND(0). However, there may be circumstances where it must be used; for example, to increase speed (see the timing results below). To make RND(0) truly random, the stopped timer must be started and kept running. This short program, for the C128, will start the timer, and set it counting down repeatedly from 65535.

```
300 bank 15
310 sa=56320:poke sa+4,255:poke sa+5,255
320 poke sa+14, 1: end
```

Re-run the previous RND(0) examples, after starting the timer, and you will see that RND(0) is now well-behaved. Unfortunately, any serial port operation, such as calling up a DIRECTORY, will stop the timer until the program above is run again.

The C64 version of RND(0) is quite similar (see Listing #2), but uses some different registers on CIA #1 to generate the random number; the two registers of Timer #A, for Bytes #1 and #3, and the 1/10 second and seconds registers of the Time of Day Clock, for Bytes #2 and #4. To watch these registers on the C64, run this program:

```
150 sa = 56320
160 print peek(sa+4), peek(sa+5),  peek(sa+8),
      peek(sa+9)
170 go to 160
```

Note that the last two registers are fixed at zero; thus, RND(0) on the C64 is even more limited than the C128 version. Run the program found in Line 200 above, to demonstrate the effect of this problem on RND(0).

The Time of Day Clock is running; however, until a write to the 1/10 second register takes place, the registers are latched. Run this program:

```
180 poke 56328,0
```

to allow the registers to follow the time; re-run the program at Line 150 and see the change.

Unfortunately, this does not improve RND(0) on the C64 very much. The 1/10 second register only runs up to 9 (decimal), and the seconds register only runs up to 89 (decimal), $59. So even if the Time of Day Clock registers are changing, the random numbers generated are still clustered about 256 values.

Another problem with RND(0) is that, if it is used repeatedly, in succession, in a BASIC program, the timers will change in a predictable way in the constant interval between successive calls to RND, yielding *pairs* of random numbers that are highly correlated.

For example, using RND(0) twice to simulate rolling a pair of dice will produce a very unusual, non-random distribution of pairs of values; some pairs of values will never appear! Try this short program:

```
400 dim a(6, 6)
410 x = rnd (.) : y = rnd (.)
420 x = 6*x+1 : y = 6*y+1
430 a(x, y) = a(x, y) + 1
440 get q$: if q$="" then goto 410
450 print:print
460 for k = 1 to 6
470 for l = 1 to 6
480 print a(k, l),
490 next l : print : next k
500 go to 410
```

Press any key occasionally to see an update of how often each possible pair of values shows up. Note the extreme non-random distribution of the pairs of values. Switching to FAST mode accentuates the problem. Interrupts that occur between the two calls to RND(0) allow the timers to run for different lengths of time, and increase (slightly) the randomness of the distribution of pairs of values. The results of this program on the C64 are still distorted, but less severely.

For both these reasons, RND(0) should be used only to create new values for RNDX, as was the case of for RND(-)

RND(+)

(The C128 and C64 versions of this part of RND are identical.)

Referring once more to the disassembly of RND, if the argument is positive, the routine branches to $8455. Here, the number stored in $121B - $121F (RNDX) is recovered into FPA #1, and multiplied by the constant stored (in floating-point form) in ROM at $8490 - $8494. The constant in $8495 - $8499 is then added. The routine then falls through to the code that juggles the bytes, sets the sign and exponent, normalizes and finally stores the new value back in $121B.

Note that the value found in FPA #1 at the beginning of the routine, the positive argument of the RND function, is never used. Instead, it is over-written by the last random number, stored in RNDX. It is this old random number that is used to create the next random number, which in turn becomes the seed for the next random number.

Any positive argument, constant or variable, for RND would have produced exactly the same new random number from a given seed.

From this description, it is clear that, given a particular value stored in RNDX, repeated use of RND with any positive argument will give the same sequence of random numbers every time. The cold-start routine initializes RNDX to zero, so that simply using RND with a positive argument would always yield the same sequence after resetting. The only way to change to a new sequence of random numbers is to change to a new value in RNDX; use RND(-), if you want to switch to some specific sequence, or RND(0), if you wish to jump to an undetermined new sequence.

One important question about RND(+) is: how long before the random numbers generated this way begin to repeat. Remember, as long as you use RND(+), you are following a fixed path, jumping in a definite route from one number to another. Sooner or later, you must land on a number you have been to before. After all, there are only so many different numbers on which to land. Once this happens, you then repeat your previous sequence of random numbers, over and over. The length of this loop is dependent on the constants used to generate the new random number from the old one. I was curious as to how well the C64 and C128 designers had chosen these constants. The machine language program (Listing 3) was written to determine the length of these loops.

The problem of finding the loop length is complicated by the possibility of a situation like this: random seed A produces B, which produces C, which produces D, which produces E, which produces F, which produces C, and the loop is completed without ever returning to A. The sequence consists of a loop, C, D, E, F, as well as a "tail", A, B. There is not enough room to keep every random number of the sequence in memory, nor is there time to check the current random number against all previous random numbers to find a match when the loop closes.

The solution is to run two sequences. They both start with the same seed. One sequence calculates RND(+) once; then the other sequence calculates RND(+) twice; then the process is repeated. Each sequence loads its own old seed into RNDX before taking its single or double step, and saves its new current seed afterwards. Eventually the double-stepping sequence will complete the loop and catch up with the single-stepping sequence from behind. The number of steps needed is the length of the loop.

The situation is a bit more complicated if the sequence starts on the tail attached to a loop, as described above. If the tail is long, compared to the loop, then the program in Listing 3 will record a length that is approximately equal to the length of the tail. The reported length will be a multiple of the loop length.

When this ML program was run with over 1000 different starting random seeds, a remarkable pattern was discovered. Fully 83% of the sequences looped in the same length: 63,671 random numbers! About 2% of the sequences contained exactly twice as many numbers; these were determined to be loops of the same length, 63,671, entered through a tail at least as long. No sequences longer than this were found.

The remaining 15% of the sequences were considerably shorter. 5% of the sequences traced had less than 10,000 random numbers. The shortest sequence observed had only 590 numbers in it!

The short sequences showed another strange property. They almost all turned out to be loops with tails, and the loops were of only a few different lengths. For example, about 9% of the sequences studied turned out to end in loops of length 724. Some of these sequences had very long tails, twenty or thirty thousand numbers long, but the loop at the end was quite short. Another 2% of the sequences studied ended in loops of length 7036, while a few other loop lengths, like 4232, 2644, and 5660 showed up at the end of a few sequences. The reason that these loop lengths, and only these loop lengths, were observed is a mystery to me.

Despite the short loops, Commodore seems to like this particular version of RND(+). The two constants used to generate the next random number are 11879546 (multiplicand) and 3.92767774E-8 (addend). These are exactly the same constants as RND uses in the C64, and thus RND(+) on the C64 suffers from the same problem as RND(+) on the C128. To illustrate this problem, run this short program on either a C128 or C64, with the appropriate value of **sa**, the address of the start of RNDX:

```
600 sa=4635 : rem for c128, sa=139 for c64
610 bank 15 : fast : rem on c128 only
620 for k=0 to 4:read x:poke sa+k, x: next
630 data 128, 115, 153, 56, 197
640 y = rnd(1): print y
650 for k = 1 to 294: x = rnd(1) : next
660 go to 640
```

This program seeds the five bytes of RNDX with values that place it on a loop of length 295. If you now use RND(+) repeatedly, you will generate the same 295 random numbers over and over again. As you can see from the output of this program, the same value of Y is generated and printed each time around the loop. All the other 294 random numbers are repeated as well.

A curious programming trick turned up when I checked the values of these RND constants in earlier PET/CBM systems. The designers managed to store two five-byte constants in only eight bytes. They accomplished this apparent miracle by overlapping the constants. Thus, the fifth byte of the first constant is also the first byte of the second constant; the first byte of the routine is also the fifth byte of the second constant. When this routine was transferred to the more spacious C64 and C128, the constants were given five bytes each. But the two new fifth bytes were set to zero. Thus the constants were changed in value slightly, and the sequences of random numbers generated by RND(+) on the PET/CBM and on the C64 and C128 are completely different.

Although 63,671 random numbers seems like a lot, it should be considered in light of the over 35 billion possible random numbers. Clearly, for any particular starting random seed, only a small fraction of all possible random numbers is available on that particular sequence. If the total number of random numbers used exceeds 63,671, as it well might, the selection of numbers will *not* be random. There is also the distinct possibility of hitting a much shorter sequence. For important applications, it would be a good idea to use RND(-) or RND(0) frequently, to jump to another loop.

Timing

As the disassembly indicates, there are large differences in the three possible routes through the RND function, resulting in large differences in execution time. These execution times were measured for the C128 by executing a short program with a loop that executed 20,000 times. The time taken for the statement **x = rnd(y)** was compared to the time for the statement **x = y**. This eliminated the overhead time devoted to finding and storing variables. Naturally, **y** was allowed to be negative, zero, and positive. Several significant results were obtained.

First, RND(0) had the shortest execution time. RND(-) took 4.43 times as long to execute, and RND(+) took 4.69 times as long, compared to RND(0). Secondly, the time to parse and evaluate a numerical constant argument was significantly longer than the time needed to find and transfer a variable argument. For example, the loop took about 10% longer to complete using **x = rnd(-1)**, compared to **x = rnd(y)**, where **y** had earlier been defined as -1. The absolute speed champ was RND(.); the Basic Interpreter recognizes a solitary decimal point as a zero faster than it recognizes the digit zero, and faster than it can look up a variable equal to zero, even if that variable is at the beginning of the variable table.

Good advice

1) Avoid using RND(0) unless absolutely necessary. Start and re-start Timer A in CIA #1, if you insist on using RND(0) on the C128. For maximum speed, use RND(.)

2) Use RND(-) to establish a new sequence of random numbers. Use as the argument a negative number that does not have an exact representation in floating point binary notation. Avoid numbers like -78, -548, and -12.875

3) Use RND(+) for most purposes. Use a variable argument, rather than a constant argument, for a useful increase in speed. Switch to a different sequence by using RND(-) or RND(0) before the sequence begins to repeat.

Listing 1: *Commented disassembly of the C128 rnd() routine*

```
f8434  jsr $8c57   ; determine sign of argument
f8437  bmi $846a   ; branch if negative
f8439  bne $8455   ; branch if positive
------------------------------------------------
                   ; ZERO ARGUMENT ROUTINE
f843b  jsr $a845   ; set bank 15
f843e  lda $dc06   ; fill Bytes #1 and #3,
f8441  sta $64     ; FPA #1, from Timer B,
f8443  lda $dc07   ; CIA #1
f8446  sta $66

f8448  lda $dc04   ; fill Bytes #2 and #4,
f844b  sta $65     ; FPA #1, from Timer A,
f844d  lda $dc05   ; CIA #1
f8450  sta $67     ; NOTE: TIMER STOPPED!

f8452  jmp $847a   ; jump to COMMON EXIT
------------------------------------------------
                   ; POSITIVE ARGUMENT ROUTINE
f8455  lda #$1b    ; set pointers to RND seed
f8457  ldy #$12    ; in RAM at $121B
f8459  jsr $8bd4   ; routine to unpack RND seed
                   ; to FPA #1
f845c  lda #$90    ; set pointers to multiplicand
f845e  ldy #$84    ; at $8490 in ROM

f8460  jsr $8a08   ; routine to unpack constant to
                   ; FPA #2 and multiply by FPA #1,
                   ; leaving result in FPA #1

f8463  lda #$95    ; set pointers to addend
f8465  ldy #$84    ; at $8495 in ROM


f8467  jsr $8a12   ; routine to unpack constant to
                   ; FPA #2 and add to FPA #1,
                   ; leaving result in FPA #1

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

                   ; NEGATIVE ARGUMENT ROUTINE

f846a  ldx $67     ; swap Bytes #1 and #4
f846c  lda $64     ; of FPA #1
f846e  sta $67
f8470  stx $64

f8472  ldx $65     ; swap Bytes #2 and #3
f8474  lda $66     ; of FPA #1
f8476  sta $65
f8478  stx $66

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

                   ; COMMON EXIT ROUTINE

f847a  lda #$00    ; set sign byte of FPA #1
f847c  sta $68     ; to zero for positive
f847e  lda $63     ; copy exponent of FPA #1
f8480  sta $71     ; to rounding byte
f8482  lda #$80    ; set exponent byte of FPA #1
f8484  sta $63     ; to $80
f8486  jsr $88b6   ; normalize FPA #1
f8489  ldx #$1b    ; set pointers to RND seed
f848b  ldy #$12    ; AT $121B in RAM
f848d  jmp $8c00   ; JMP to routine to pack FPA #1
                   ; into RND seed; use that routine's
                   ; RTS to exit
```

Listing 2: *Commented disassembly of the C64 rnd() routine*

```
e097   jsr $bc2b      ; determine sign of argument
e09a   bmi $e0d3      ; branch if negative
e09c   bne $e0be      ; branch if positive
------------------------------------------------
                      ;ZERO ARGUMENT ROUTINE
e09e   jsr $fff3      ; set up indirect address
e0a1   stx $22        ; to $DC00 in $22 and
e0a3   sty $23        ; $23

e0a5   ldy #$04       ; fill FPA #1,
e0a7   lda ($22),y    ; Bytes 1 and 3,
e0a9   sta $62        ; from Timer A,
e0ab   iny            ; CIA #1
e0ac   lda ($22),y
e0ae   sta $64
```

```
e0b0   ldy #$08        ; fill FPA #1
e0b2   lda ($22),y     ; Bytes 2 and 4,
e0b4   sta $63         ; from Time of Day Clock
e0b6   iny             ; Registers, CIA #1
e0b7   lda ($22),y
e0b9   sta $65

e0bb   jmp $e0e3       ; jump to COMMON EXIT
---------------------------------------------------
```

(remainder of routine identical to C-128 version, except for different addresses of routines, FPA #1, and constants)

Listing 3: *random.src*

```
CB 1002 sys4000
KI 1004 .org 4864
CJ 1006 .mem
EP 1008 ;   table of variables
HJ 1010 count =1024    ; start of counter
OM 1012 rndx =$121b    ; start of rnd seed
DB 1014 mmu =$ff00     ; change bank
ON 1016 rndpos =$8455  ; positive rnd entry
EJ 1018 ;---------------------------------------------
IE 1020 start    sei            ; no interrupts
IJ 1022 ;---------------------------------------------
GJ 1024 :        lda mmu        ; store bank and
NM 1026 :        sta temp       ; go bank 15
DE 1028 :        lda #00
JN 1030 :        sta mmu
CK 1032 ;---------------------------------------------
MK 1034 :        ldx #4
LI 1036 :        lda #0
NO 1038 loop1 =*
IN 1040 :        sta count,x    ; zero counter
BE 1042 :        dex            ; all five bytes
FE 1044 :        bpl loop1
AL 1046 ;---------------------------------------------
LP 1048 :        ldx #4         ; store current rnd seed
LO 1050 loop2    lda rndx,x
PG 1052 :        sta seed1,x    ; in seed1 and seed2
BD 1054 :        sta seed2,x
PE 1056 :        dex            ; all five bytes
GF 1058 :        bpl loop2
OL 1060 ;---------------------------------------------
NF 1062 main =*
CM 1064 ;---------------------------------------------
JI 1066 :        ldx #4         ; move rnd seed to
OD 1068 loop3    lda rndx,x     ; seed2 and seed1
MN 1070 :        sta seed2,x    ; to rnd seed
MO 1072 :        lda seed1,x
GL 1074 :        sta rndx,x
DG 1076 :        dex            ; all five bytes
NG 1078 :        bpl loop3
CN 1080 ;---------------------------------------------
DP 1084 :        jsr rndpos     ; do a rnd(+)
IN 1086 ;---------------------------------------------
CO 1088 :        ldx #4
BM 1090 loop4    inc count,x    ; increment counter
DC 1092 :        bne done       ; check for carry
CP 1094 :        dex
CI 1096 :        bpl loop4
AA 1098 :        jmp exit       ; counter overflow
GO 1100 ;---------------------------------------------
MC 1102 done =*
HG 1104 :        ldx #4         ; move rnd seed
IC 1106 loop5    lda rndx,x     ; to seed1 and seed2
AA 1108 :        sta seed1,x    ; to rnd seed
EB 1110 :        lda seed2,x
PF 1112 :        sta rndx,x     ; all five bytes
GA 1114 :        dex
JJ 1116 :        bpl loop5
IP 1118 ;---------------------------------------------
HI 1120 :        jsr rndpos     ; do rnd(+)
NA 1122 :        jsr rndpos     ; twice
OP 1124 ;---------------------------------------------
IF 1126 :        ldx #4         ; compare rnd seed
PJ 1128 loop6    lda rndx,x     ; and seed1
OD 1130 :        cmp seed1,x
LJ 1132 :        bne main       ; not equal, so start over
KB 1134 :        dex
IF 1136 :        bpl loop6      ; compare five bytes
MA 1138 ;---------------------------------------------
IN 1142 exit     lda temp       ; all five match, so
BL 1144 :        sta mmu        ; restore old bank,
PN 1146 :        cli            ; allow interrupts, and
OI 1148 :        rts            ; back to
IB 1150 ;---------------------------------------------
FH 1152 seed1 =*
GA 1154 .byte 0,0,0,0,0
LH 1156 seed2 =*
KA 1158 .byte 0,0,0,0,0
LO 1160 temp =*
OH 1162 .byte 0
```

Turning Off Write/Verify

Modifying 1571 vectors

by Dennis J. Jarvis

Before I get into the actual operating system I would like to give some background on the 1571's operating system.

When Commodore released the 2031 disk drive, which was a single drive unit, it included Disk Operating System 2.6. DOS 2.6 thus became the standard for single disk drive units. This drive is what the VIC-1540 was based on. The IEEE bus was reduced to a serial bus, and the disk formating method was changed (GAP1 was increased by 1 on the 1540).

This is why the 4040 and 2031 disk drives are *read* compatible with the 15xx series of disk drives but are *not write compatible*. For more information on this read/write problem consult the book *Inside Commodore DOS* by Richard Immers and Gerald G. Neufeld on page 208.

When Commodore released the Plus/4 computer they included a serial bus for the existing serial bus devices, but they also included a new type of bus called the TED bus. This bus was a cross between the serial bus and an IEEE bus with 8 data lines and a few handshake lines. When Commodore released the 1551 disk drive (also known as the 488 disk drive) they (David Siracusa) made several changes to the operating system, which included a faster GCR to binary conversion, a fast format routine, and corrections for some old bugs in the block-read and block-write routines.

Oddly enough, this DOS was released as DOS 2.6, which is the same DOS version that is still in the 154x disk drives, even though their memory maps are nowhere near the same. Commodore's logic in numbering their DOS versions is still not very clear to me!

Commodore based the new 1571 disk drive on the 154x's memory map. What David did was to remove a lot of the code he installed in the 1551 disk drive and splice it into the 1571 (such as the new GCR routines and the faster formating code, etc.)

I only wish Commodore had released the 1571 with a serial bus and an IEEE or 1551 type of bus which would have increased the overall speed of the drive - if the DOS was recoded correctly. To wrap up all of the above, the only drives that are both read and write compatible are the 4040 and 2031 disk drives, and the 15xx series of drives are read and write compatible. Of course, you can't put a 1571 disk in a 15xx and read side 1 since you don't have the read/write head to do it with. You can read the same disks with the 15xx, 2030, and 4040, but do not write back to them or you may regret it.

One of the changes made in the 1571 DOS was the method by which it handles the IRQs (Interrupt ReQuests). Instead of always jumping to a constant memory location (address), it jumps through a vector the same way that your computer's Kernal jumps through the ICHROUT vector to output a character to the screen/printer, etc. Since this is the first disk drive to have this ability, such programs as this were not possible before without a major effort on the part of the programmer. With this new vectoring system in your disk drive we have many options to pursue - such as the one covered in this article.

The drive's verification routine

Whenever one of the current disk drives (from the 2030 through to the current 1571) writes a sector to disk, it will then proceed to read that sector back in off the disk to verify it against what it has in RAM to ensure that the bytes were written correctly.

Generally you will never see an error occur unless you have a bad diskette or a hardware failure in the disk drive or serial bus. Normally a bad disk is detected during the format process, but they do fail from time to time just by sitting in your disk box.

In either case, you won't see the problem very often. What this program does when installed in the 1571 is to 'wedge' itself into the IRQ vector to allow my program to check to see if the DOS is about to do a VERIFY operation; if it is, I replace it with a SEEK operation. If you don't understand any of this, don't worry about it as it is not required to know *how* the program works to use it.

This program could be considered risky to some degree and it probably *is* risky, but I've been using this program daily for three months with *no* problems encountered to date. Do I

This program runs on any Commodore computer from the VIC-20 to the C128...

guarantee this program? *Not on your disk I don't.* Why, you might ask? Well, to put it simply, there are too many variables involved; from your disk drive having the flu, to disks that are used as floor mats. It would only take one smudge to wipe a disk. On any account, simply run this program and give it a try by doing some testing on a *junk disk.* (You know the one, it's the one you don't care if your dog or cat eats, or if Junior uses it as a frisbee! What I'm trying to say is that I've used this program for quite a while with *no* problems on several different versions of the 1571 Disk Operating System, but this choice is up to you to make, not me.)

This short program allows you to turn off the write verify operation in your 1571 disk drive even while it's in the 1541 mode. Currently there are several similar versions of this program running around on public domain bulletin boards but several of them contain bugs because they, like this one, sit some where in $0100, which is the disk drive's stack. Most of the ones I have seen to date use up too much room on the stack and, upon the first DOS error, - crash! - the drive dies. The following program has been tested, tested again, then tested more...

To use this program all you need is a 1571 disk drive and this program. This program runs on *any* Commodore computer from the VIC-20 to the C128. To start the program, just load it up and type **run**. The program will make only one request, and that is, simply: "Enter the device number of your 1571 disk drive:". This value can be in the range of 5 through 30, and would be the same number that you would enter for a BASIC command as **dload"filename",u(*xx*)** or **load "filename",*xx*** where ***xx*** is the device number of the 1571 disk drive. If the program finds that you indeed have a 1571 disk drive at that device number, it will proceed with its assigned task. If there is *not* a 1571 online, it will display an error message. When the program has turned off the write verify operation it will inform you by displaying "Write verify operation is now turned off!"

During the course of this program I make several checks to ensure this is really a 1571 disk drive: First I check the IRQ vector at $FFFE to ensure it contains an $FE67; this will ensure that it is a 154x or 157x type of drive. Next I check $8002 for the text **S/** to ensure it is a 1571 disk drive. Finally I check $02A9 which is the IRQ vector for the 1571 disk drive. When I check this vector I ensure that the MSB of the address contains a value greater than $80 to ensure that the IRQ is going somewhere into ROM, and not to another program such as mine. If all of the aforementioned are true, I download the ML into the drive and execute it.

"verify.bas" - *Turns off write verify on the 1571*

```
FP  10 rem publication rights reserved
HM  20 printchr$(142)"{home}{home}{clr}{down}{down}{down}{down}{down}{down}
         {white}turn off write verify operation for"
NM  30 print"     commodore's {cyan}1571{white} disk drive
DA  40 print"{down}     {cyan}(c){white} 1988, 1989
CA  50 print"{down}                by
HN  60 printchr$(15)"{down}     dennis j. jarvis"chr$(143)
JO  70 print"{down}     kissimmee florida"
KE  80 printchr$(7)chr$(15)"{down}{down}  {lt. blue}press any key to
         {space}continue"chr$(142)
BH  90 geta$:ifa$<>""then90:rem purge buffer of any previous key strikes
JJ  100 geta$:ifa$= ""then100:rem wait for a key to be pressed now
EH  110 print"{clr}"
DG  120 input"{white}what is the 1571's device number  8 {left}{left}
         {left}{left}";a$:dv=val(a$)
JG  130 ifdv<5ordv>30then120
CJ  140 print"{clr}"
MD  150 open15,dv,15
DP  160 close15:ifstthenprint"{clr}{down}{down}{down}{down}{right}{right}
         {right}{right} device number:"str$(dv)" is not turned on":end
AF  170 open15,dv,15
HF  180 lsb=254:msb=255:gosub350:rem read the rom irq vector ($fffc)
BO  190 iflsb=103andmsb=254then220
FL  200 print"{clr}{down}{down}{down}{down}{down}{right}{right}{right}{right}
         {right}{right}sorry this program will only work on the 1571 disk drive"
DL  210 close15:end
ND  220 lsb=002:msb=128:gosub350:rem check text at $8002 's/w - david
         g. sir...etc
DC  230 iflsb<>83ormsb<>47then200
GG  240 lsb=169:msb=002:gosub350:rem make sure irq is pointing to a routine in rom
DK  250 ifmsb<128thenprint"{clr}{down}{down}{down}{down}{down}{space}
         sorry some one is using the irq vector":goto210
OK  260 fori=2to50:read h$:a$=left$(h$,1):b$=right$(h$,1)
CH  270 gosub300:b=a*16:a$=b$:gosub300:c=a+b
FB  280 print#15,"m-w"chr$(i)chr$(1)chr$(1)chr$(c):next
BB  290 print#15,"m-e"chr$(23)chr$(1):print"{clr}{down}{down}{down}{down}{down}
         {right}{right}{right}{right}{right} write verify now turned off":end
JL  300 ifa$<":"anda$>"/"thens=48:goto330
CA  310 ifa$>"@"anda$<"g"thens=55:goto330
GE  320 print "invalid hex byte -"h$":stop
FA  330 a=asc(a$)-s
AH  340 return
GH  350 print#15,"m-r"chr$(lsb)chr$(msb)chr$(2):get#15,a$,b$:lsb=asc(a$+chr$(0))
         :msb=asc(b$+chr$(0)):return
KJ  360 data  48:          :rem pha         save acc.
JK  370 data  8a:          :rem txa
PB  380 data  48:          :rem pha         save x reg. onto the stack
NM  390 data  a2,05:       :rem ldx #5      number of job queue's to check
GK  400 data  b5,00:       :rem lda $00,y   get current job out of queue
EF  410 data  c9,a0:       :rem cmp #$a0    is it a verify command?
OF  420 data  d0,04:       :rem bne *+15    no then branch
DF  430 data  09,10:       :rem ora #$10    replace the verify function
LB  440 data  95,00:       :rem sta $00,y   with a seek command
OL  450 data  ca:          :rem dex         move down to next queue
NC  460 data  10,f3:       :rem bpl *-11    if any more to do branch
BE  470 data  4c,fe,ff:    :rem jmp $0000   see text for this one
EF  480 :
DO  490 :rem this is where the program is executed when it's in the disk drive
MG  500 :^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
MO  510 data  78:          :rem sei         stop background jobs
NE  520 data  ad,a9,02:    :rem lda $02a9   get the address of the
LE  530 data  18:          :rem clc         current addres in the irq
AJ  540 data  69,03:       :rem adc #3      vector and add 3 to is then
NI  550 data  8d,15,01:    :rem sta $0115   save it for us to jump to,
AO  560 data  ad,aa,02:    :rem lda $02aa   this was done to save code
DP  570 data  8d,16,01:    :rem sta $0116   (place at jmp $xxxx
JM  580 data  a9,02:       :rem lda #2      place the address of our
BB  590 data  8d,a9,02:    :rem sta $02a9   new irq routine into the
CO  600 data  a9,01:       :rem lda #1      the 1571's irq vector
CG  610 data  8d,aa,02:    :rem sta $02aa   so were called each irq.
LO  620 data  58:          :rem cli         restart the background jobs
GL  630 data  60:          :rem rts         and were done
```

Make 2 Sided

Converting 1541 disks to 1571 format

by Dennis J. Jarvis

This program was written to solve two problems I was having with the 1571 disk drive:

1) Converting a single-sided diskette to a double-sided one

2) Doing a COLLECT on a 1571 disk while it was in 1541 mode, causing it to revert to being a single-sided disk. (**NOTE:** This problem has been corrected in the newer 1571 disk drives.)

The first problem can be a pain in the keyboard. What was required was that I use the DOS shell or other such program and copy the files over to a newly formated 1571 diskette. I could not use any full disk copiers because they would copy track 18 and 0 from the single-sided disk to the double-sided disk, turning it back into a single-sided disk (as far as the 1571 was concerned). Well, that got to be a time-consuming operation. So I set out to find an easier way. Using my *1571 Internals* book from Abacus Software, I began to scan through the format routines. As it turned out, when David Siracusa wrote the format routine, he set it up as a two-pass operation. First, he formats all of side 0, then returns to format side 1.

Before I go into any further details on how to use the format routine I would like to tell you how the 1571 checks to see if it's got a double-sided diskette in the drive or not. When you perform most disk functions, the Disk Operating System (DOS) reads track 18 sector 0 into the disk drive's RAM, then checks the third byte into that sector (the first byte is byte 0) and checks to see if it has bit 7 set. If it is *not* set, it is a single-sided disk. If bit 7 *is* set, the drive will read in track 53 sector 0 to read the rest of the disk's Block Availability Map (BAM). This information is used to find out if a sector on a specific track on side 1 of the disk is in use or if it is free.

As you can see, before the DOS will allow you to read a track beyond track 35 using the normal DOS, it needs to see bit 7 set in the third byte of track 18 sector 0. My program will always set this bit. After this has been done I send out an **I0:** which does a couple of things: First, it forces the disk drive to reload track 18 sector 0 into its RAM (the BAM for side 0) and set the internal single/double-sided disk flag. Second, it will force the DOS to read in this disk's formatting IDs.

Whenever you format a disk using a statement such as **"n0:commodore 1571,dj"** it will make the disk's name **commodore 1571**, and place the formatting IDs in each header preceding each sector. In this case the formatting IDs would be **dj.** These ID bytes are placed along with the name on track 18 sector 0. These can be changed by various software programs such as *HEADER CHANGE*, which is on the 1571 test/demo disk which was bundled with your 1571. If you change the name of the disk or the disk ID on track 18 sector 0, you are just changing the information that you would see when you load/read in the directory. The actual formatting ID bytes cannot be changed without reformatting the disk from scratch.

Once the disk has been set up as a double-sided disk, and has read in the disk ID bytes, we're ready to go to work. Next, I set the read attempts to one to prevent the disk from searching too long (so it does not waste your time), then I attempt to read track 53 sector 0. Track 53 sector 0 could have been any track on the second side of the disk. If I'm able to read in this track and sector, then I know that the disk was previously formatted as a double sided disk, but has reverted to a single-sided disk due to problem two stated above. All I have to do is to perform a **collect** command on the disk to reconstruct a valid (correct) BAM. If I'm unable to read track 53, sector 0 then I know that side one was never formatted, and I need to format it.

Note that this program does *not* convert flippies. Flippies are those disks that have been notched on the other edge so that they can be flipped over and used on the other side. In order to format side one only, all I have to do is to enter the formatting routine at $A445 in the 1571 disk drive. Once this is done the disk drive will go on its way formatting side one *only* of this disk. If no errors occur, performing a **collect** on the disk will create a BAM for the second side. Once this has been completed you will have a double-sided disk with no files on side one.

Even though this program may sound straightforward, *it is not!* You must allow *make 2 sided* to complete its task or it will cause you problems in the future. An example: you have a single-sided disk that you want to convert to a double-sided disk. You get all the way up to the point where the disk was just about to be formatted (side one only, of course), and you change your mind and pull the disk out of the disk drive. If you do not rerun this program and select **no** (when it asks you if you want to format the disk), you will find that you have a big problem on your hands...

Since my program has already set the double-sided flag so that I could read side one of the disk, the next time you try to do anything with that disk (such as saving or loading a program) the 1571 sees the double-sided flag set and tries to read track 53, sector 0 to read in the rest of the BAM and - bang! - a read error because there is *no track 53, sector 0* because it was never formatted!

To use *make 2 sided* all you need to do is to type it in or obtain the *Transactor* disk for this issue *[Tdisk #27]*, run it, and then answer the questions. They are very straightforward questions such as entering the device number of your 1571. This program has been tested and operated without any problems on a VIC-20, C64, Plus/4, B-128 (with our fast serial bus installed, of course), and the C128. I made sure that I only used BASIC 2.0 commands to allow everyone that owns a 1571 a chance to use this program.

Make 2 sided has been tested, tested, tested, then tested some more. This program has already been released via the CBUG user's group for the B-128 computer with no complaints thus far.

As with any piece of software, there is room for change and growth, but this is a functional program without any bells and flags. If you wish to change this software, make sure you know *exactly* what you are doing or you could erase the wrong side of the diskette or find yourself with a new dent in your 1571's outer case because you told your disk drive's head to take a quick trip to Mars and back.

When *make 2 sided* was sent in, it was over four pages long - mostly because of single line statements, and massive amount of comments. Probably the comments will be removed, and the program shortened due to space limitations. I hope that *Transactor* will place the BASIC program on their diskette as it was sent to them so you can obtain a copy to follow exactly how this program works. [Mr. Jarvis' program has not been shortened. In fact, a line was added to get around the appearance of an ELSE token. Aside from that, nothing has been removed. - MO]

A couple of closing comments about track 18 sector 0: If you look at the last few bytes starting at byte $DD you will see the number of sectors free on each track on side one. I'm not certain but, in my opinion, David did this to allow us (programmers) a way of seeing if any tracks are in use on the other side of the disk.

When I wrote a full disk copier for the CBUG (B-128 users group) I used this method to see quickly if any sectors were in use on side one of the disk. This would give them the option of using a 154x, 2031, or 4040 disk drives as the source disk and the 1571 disk drive as the copy disk. The only drawback I can see (and have been affected by) is that my normal method of reading in a disk's BAM and directory no longer works.

If you open the directory up for *reading* (not **load"$",8** etc.) you will be able to read in the disk's BAM at the same time. On the 1571 it will *not* send you the BAM for side one; you have to go get it yourself with a U1 command. One last comment about the format ID byte in the BAM: This byte (byte 2 starting with byte 0) is a $41 on the 4040, 2031, and 15xx drives. This byte has been confused over the years as the format type byte. In my *opinion* this tells you the number of sectors per zone and is laid out as follows:

```
DRIVE      ZONE        FORMAT TYPE
        1  2  3  4
------------------------------
2040   20 19 17 16    $31 or $A0
3040    "  "  "  "        "
1540   20 18 17 16      $41
1541    "  "  "  "        "
1551    "  "  "  "        "
1571    "  "  "  "        "
2031    "  "  "  "        "
4040    "  "  "  "        "
2030   (DO YOU KNOW?)   $42
8050   28 26 24 22      $43
8250    "  "  "  "        "
```

All of the above table is of my own creation and opinion. Take the FORMAT TYPE $41 for example. On the 154x and 1551 they are all read/write compatible and are identical in about every way and are currently formatted exactly the same. On the 4040 and 2031 drives, there is an extra GAP1 byte in the format process when compared to the 15xx drives. Due to this extra byte, they are not write compatible with the 15xx.

The 1571 is double-sided drive with a different BAM scheme when compared to that of the 2031, 4040 and 154x, and 1551 type drives. In other words, this byte does *not* indicate the number of tracks on a disk, nor the formatting method used on the disk as you are led to believe in the disk drive's manual.

One interesting point: on all of the Commodore disk drives, the DIRECTORY routine defaults back to the 2040 disk drive which was VERSION 1 as the format type. On the 4040 if you change the 2A on track 18 sector 0 to 2 followed by an $A0 it will show it as 21, after the disk name!. This holds true for all on the Commodore disk drives, including the 8050/8250 drives as well!! (change the 8050, 8250 from 2C to 2, followed by an $A0).

If anyone has any input on my opinion, as stated above, please send it into *Transactor*, I would like to hear from such people as Fred Bowen, Jim Butterfield, Liz Deal, Jesse Knight, Anthony Goceliak, David Siracusa or from anyone wanting to make a comment! All of the aforementioned programmers have been programming on various Commodore disk drives for many years, and have far more hands-on time than I do. Like so many others, I'm still just a beginner.

Hopefully, in the next month or two I will have found the time to clean up the comments in my source code for my *Fast Format* for the 154x, and 1571 (in 1571, or 1541 mode), and will be able to get them off to *Transactor* - and to you. Until then, keep those disk drives spinning.

"make 2 sided.bas"

```
MF  10 rem ***********************************************************************
IF  20 rem *                                                                     *
HA  30 rem *    this program is used to convert a double sided diskette that     *
FG  40 rem * was collected on the 1571 disk drive while it was in the 1541 mode  *
CN  50 rem * which will automaticaly convert a diskette back to a single sided   *
EE  60 rem * diskette. this progam will also convert a diskette formatted        *
AF  70 rem * on a 1541 to a double sided 1571 diskette.                          *
EI  80 rem *    to use this program just place the affected diskette            *
OI  90 rem *  in the 1571 disk drive and run this program, and if this          *
LK  100 rem * problem holds true then this program will correct the problem.     *
CL  110 rem *                                                                    *
EG  120 rem * warning: if for any reason any errors occur or if during the       *
JF  130 rem *          execution of this program you should change your mind    *
PK  140 rem *          just answer (n)o to the questions, if for some reason    *
BI  150 rem *          (such as a write error or a power failure) this program  *
LG  160 rem *          does not finish its task you must rerun this program or  *
DM  170 rem *          serious problems could occur with your diskette!.         *
IP  180 rem *                                                                    *
DM  190 rem *  (c) 1988, 1989   by dennis j. jarvis  publications rights reserved*
MA  200 rem *                                                                    *
EC  210 rem ***********************************************************************
DL  220 a=40 :rem  change this to your computers screen size 40, 80 etc.
OD  230 sc=a/2:printsc
FB  240 print"{home}{home}{clr}":rem clear the screen and delete any
                                current windows (if any)
EI  250 dv=8
LM  260 p$="enter the disk drives device number":rem string to be printed
EF  270 row=12              :rem row to print the string on
JE  280 gosub1820           :rem print it to the center of the screen
BK  290 printdv;"{left}{left}";    :rem display the default drive number
EA  300 open5,0             :rem prevent question mark from being printed
EM  310 input#5,a$          :rem allow the user to change the current device number
GE  320 close5              :rem allow proper screen prompting
EJ  330 dv=abs(val(a$))     :rem make the device number positive
CD  340 ifdv = 0 then print"{clr}":end:rem they're done now
CM  350 ifdv>5  and  dv<31 then 430: rem if the device number is in range branch
HN  360 print"{clr}"      :rem if it's not then clear screen and print the message
FP  370 p$="illegal device number " :rem to the screen
FP  380 rv=1                :rem set the reverse field on flag
AP  390 row=12              :rem print it in the center of the screen
JH  400 gosub1820           :rem print it to the screen
OJ  410 gosub1700           :rem wait for the user to acknowledge the error
HM  420 goto250             :rem restart the input loop
PN  430 open1,dv,15,"u:" :rem soft reset the disk drive to obtain model number
PE  440 forx=0to2000:next:rem give the drive time to finish its reset process
JN  450 gosub2050           :rem read in and save disk drives model number
BP  460 print#1,"u0>m1"  :rem just in case were running on a non fast bus computer
CF  470 print#1,"u0>r"+chr$(1)
NC  480 gosub2080           :rem close down the command channel to the disk drive
DB  490 ifleft$(right$(e$,9),1)="7"then o=1:goto510
IG  500 o=0
HC  510 ifothen570        :rem if it's a 1571 online then bypass the error message
FO  520 print"{clr}":p$= "sorry for the 1571 disk drive only"
          :rem string to be printed
KH  530 row=12            :rem print it on this row
CC  540 rv=1              :rem print it in reversed field
HF  550 gosub1820         :rem print it to the center of the screen
NL  560 gosub2080:forx=0to3000:next:run
EH  570 gosub1660               :rem open the proper channels to the disk drive
PK  580 f1=128:gosub1230        :rem set the flag for double sided disk
IG  590 print#1,"u0>r"+chr$(1):rem set read attempts to 1
ON  600 print#1,"u1:2 0 53 0" :rem attempt to read second side of the disk
FN  610 gosub2050:d=e           :rem preserve the error channel
JP  620 print#1,"u0>r"+chr$(5):rem reset number of read attempts
GD  630 if d = 66 then 810      :rem if unable to read the second side of the disk
MB  640 rem                           then branch to see if they want to format it
FE  650 rem                           d will equal 66 ( illegal track and sector if
CC  660 rem                           it's unable to read the second side of the disk)
KB  670 gosub2080       :rem close all open channels to the disk drive
JC  680 gosub2090       :rem make the drive recognize that this disk 2 sided disk
                             and have it read in the disk id's
GJ  690 print"{clr}
MG  700 p$="disk has now been restored to a   double sided diskette"
BN  710 row=11:gosub1820
FP  720 p$="i am now performing a collect on  this disk please wait..."
DL  730 row=13:rv=1:gosub1820:gosub2080:gosub2090:open15,8,15,"v0:":close15
          :print"{clr}":end
AE  740 rem *-----------------------------------------------------------------*
HO  750 rem *  if we are unable to read the second side of the diskette then we  *
OH  760 rem * give the user the option of formatting only the second side of the *
GM  770 rem * disk. if selected then we will format only the second side of the  *
DC  780 rem * disk, or if not wanted then we restore the single sided flag on    *
CM  790 rem * the disk and terminate this routine.                               *
MH  800 rem *-----------------------------------------------------------------*
OA  810 print"{clr}
MD  820 gosub2080
CA  830 p$="sorry this diskette was not formatted as a double sided diskette."
BB  840 row=10:gosub1820:p$=" would you like to make it one? (y/n) ?":row=12:rv=1
          :gosub1820:gosub1920:
GD  850 print"{clr}
KP  860 p$= "are you sure? this will erase the second side of your disk? (y/n)"
AA  870 row=13
PI  880 gosub1820
LP  890 gosub1920       :rem get their response if it's 'y' then return
GA  900 gosub1410       :rem format the second side of the diskette
JJ  910 goto690         :rem inform the user and perform a collect on the diskette
BL  920 gosub2080       :rem close down all open files
MN  930 gosub1660       :rem open the proper channels to the disk drive
HO  940 f1=0            :rem set the flag for a single sided diskette
NP  950 gosub1230       :rem reset single sided diskette flag
MN  960 return
GC  970 rem *-----------------------------------------------------------------*
II  980 rem *   this routine is the error handler. if any disk errors have       *
HH  990 rem * occured ,other than the drive being off ,the routine outputs the   *
FJ  1000 rem * error message and resets the double sided/single sided diskette   *
EO  1010 rem * back to a single sided diskette. note that this is very important!!*
CC  1020 rem *----------------------------------------------------------------*
FI  1030 if (st and 128) then1130 :rem check for the disk drive being turned off
OH  1040 gosub2050:ife<2 then return  :rem if there are no drive errors then return
BN  1050 print"{clr}":open5,0
JJ  1060 p$= "disk error has occurred - ":row=11:r=0:gosub1820:gosub2050:p$=e$
           :row=13:r=1:gosub1820:print:goto920
EF  1070 rem *----------------------------------------------------------------*
IC  1080 rem *   this routine is used by the trap command to detect such errors  *
OC  1090 rem * as syntax, or the disk drive being turned off, if the disk drive  *
OF  1100 rem * is turned off then it prints a warning to the screen and restarts *
OO  1110 rem * the program. if any other error occurs then the program is aborted.*
GI  1120 rem *----------------------------------------------------------------*
CO  1130 p$=" turn on your disk drive ":rv=1:row=12:gosub1820:gosub1700:run
KJ  1140 rem *----------------------------------------------------------------*
MI  1150 rem *  read the diskettes bam into the buffer and set the flag to       *
```

```
EM 1160 rem * indicate this disk is double sided to enable us to read the second *
AK 1170 rem * side of the disk, and if we are able to read it then this diskette *
GM 1180 rem * was formatted as a double sided disk, but if we are not able to    *
OP 1190 rem * read the second side of the diskette then it was formatted as a    *
LI 1200 rem * single sided diskette so reset the flag back to indicate that this *
EA 1210 rem * is a single sided diskette and end this program.                   *
KO 1220 rem *-------------------------------------------------------------------*
CL 1230 print#1,"u1:2 0 18 0" :rem read in the diskettes bam
PD 1240 gosub1040             :rem check for read errors
JF 1250 print#1,"b-p 2 3"     :rem set the pointer to the double side indicator
KI 1260 print#2,chr$(f1);     :rem set the flag to the value for single
                                   or double sided
CC 1270 print#1,"u2:2 0 18 0" :rem write the new bam back to the diskette
KI 1280 gosub1040             :rem check for write/read errors
BC 1290 print#1,"i0:"         :rem force drive to read it's new bam in
II 1300 gosub2080             :rem close down our channels (1571 just did!)
MA 1310 gosub1660             :rem reopen the channels
EE 1320 return
GK 1330 :
GC 1340 rem *     this routine is used to format the second side of the disk    *
AM 1350 rem * that is currently formatted as a single sided disk. to do this a   *
MI 1360 rem * routine that is in the 1571 disk drive is used, which is in the    *
CG 1370 rem * normal formatting process, to use this routine we place the max.   *
PH 1380 rem * number of tracks on this disk (71), into $02ac, and jump to the    *
NL 1390 rem * format routine ($a445) to format the second side of the disk       *
OJ 1400 rem *-------------------------------------------------------------------*
FL 1410 a = 284          :rem maximum track (located in the 1571's memory at $02ac)
LA 1420 b=71             :rem maximum track number-1 to format too
MA 1430 open1,dv,15      :rem open the command channel to the disk drive
GN 1440 gosub 1570       :rem check for any errors
AL 1450 print#1,"m-w"+chr$(a and 255)chr$(a/255)chr$(1)chr$(b):rem set max trk
IJ 1460 print"{clr}
FJ 1470 p$="the formatting process is now being performed on side 1 of
        {space}your diskette"
CP 1480 row=11:gosub1820:print#1,"m-e"chr$(69)chr$(164):gosub1570:close15
        :gosub2080:return
IP 1490 rem *-------------------------------------------------------------------*
FI 1500 rem *   this routine will check to ensure that there are no errors during*
IP 1510 rem * the format process, such as the user popping the diskette out of   *
ME 1520 rem * the disk drive, or any write errors, etc. if any occur then this   *
OE 1530 rem * subroutine will terminate the formatting process and reset the     *
GG 1540 rem * single/double sided flag back to a single sided diskette.          *
DF 1550 rem * note: read the warning in the first set of rem statements.         *
OD 1560 rem *-------------------------------------------------------------------*
PF 1570 gosub2050:ife<20then return :rem if no disk drive errors have
                                         occurred then return
PD 1580 print"{clr}":close5:p$="a disk drive error has occured ":row=9:rv=0
        :gosub1820:p$=e$:row=11:rv=1:gosub1820:p$="please check your disk drive "
CH 1590 row=13:gosub1820:gosub1700:gosub920:goto810
GG 1600 rem *-------------------------------------------------------------------*
KJ 1610 rem *   this routine will open the command channel to the disk drive    *
DH 1620 rem * whose number is specified in dv, and will also reserve             *
HB 1630 rem * a buffer for our use inside of the disk drive to allow us to read  *
MI 1640 rem * and write the bam from/to the disk drive                           *
IJ 1650 rem *-------------------------------------------------------------------*
FI 1660 close2:close1 :rem ensure channels are closed
MP 1670 open2,dv,2,"#":rem allocate a buffer for our use in the disk drive
GA 1680 open1,dv,15    :rem open the command channel to the disk drive
AD 1690 goto 1570      :rem check for any disk errors then return
KE 1700 p$= " press any key to continue ":rv=1:row=row+2:gosub1820:gosub2040
AN 1710 geta$:ifa$=""then1710
HN 1715 print"{clr}":return
ON 1720 rem *-------------------------------------------------------------------*
CJ 1730 rem *  this routine is used to print our text centered on the 40 or 80  *
FE 1740 rem * column text screens.                                               *
KB 1750 rem *                                                                    *
NK 1760 rem *  to use this routine you must predefine the following variables.   *
OC 1770 rem *                                                                    *
FC 1780 rem *  p$  = the string you wish to print centered onto the screen       *
LO 1790 rem *  r   = reverse field on (1), or off (0)                            *
HJ 1800 rem *  row = row number to  print the text on                            *
ID 1810 rem *-------------------------------------------------------------------*
PB 1820 nl = len(p$)/2 :rem find the true length of the string
KB 1830 print"{home}":forx=2torow:print:next
LI 1840 fori=1toabs(sc-len(p$)/2):print"{right}";:next:ifrv=1thenprintchr$(18);
BO 1850 printp$;:rv=0:return
KG 1860 rem *-------------------------------------------------------------------*
FJ 1870 rem *   clear the keyboard buffer of any key presses and wait for the   *
CK 1880 rem * user to press a key and if it's the key 'y' then return to the     *
IB 1890 rem * calling routine, else reset the flag in the bam to indicate this   *
AL 1900 rem * disk is a single sided disk, not a double sided diskette then end  *
MJ 1910 rem *-------------------------------------------------------------------*
FP 1920 gosub2040         :rem purge the key board buffer of any
                               outstanding key presses
GJ 1930 getz$:ifz$=""then1930  :rem wait for the user to press a key
EJ 1940 ifz$="y"then return
OC 1950 gosub920          :rem set the flag for a single sided diskette
GJ 1960 gosub2080         :rem close any open disk drive files
LO 1970 print"{home}{home}{clr}"     :rem clear the screen and terminate the window
BF 1980 end               :terminate this program
MO 1990 rem *-------------------------------------------------------------------*
GA 2000 rem *  generic purge routine for all cbm computers. this subroutine will *
GK 2010 rem * remove all characters entered up to this point.                    *
KA 2020 rem *-------------------------------------------------------------------*
LD 2030 getzz$:ifzz$<>""then2030   :rem any more keys in the buffer? if so loop
OL 2040 return                     :rem buffer now purged
GC 2050 e$=""
HA 2060 ifstthen e=val(left$(e$,2)):return
BE 2070 get#1,a$:e$=e$+a$:goto2060
GM 2080 close2:close1:return
EN 2090 open1,dv,15,"i0:":close1:return
```

Customizing C128 CP/M

Patches for CPM+.SYS

by M. Garamszeghy

C128 CP/M mode relies on a RAM-based operating system to control the basic functions of the computer. These RAM-based operating systems have several advantages over ROM-based systems (such as the standard Commodore 8-bit Kernal and BASIC systems) in that it is very easy to make changes, modifications and general customization by patching the RAM, whereas an EPROM burner is generally required to make changes to a ROM-based system. This article deals with a number of fairly simple, but quite useful modifications, or patches, that can be made to the CP/M boot program.

Before we get started, a bit of background info may be useful. Officially, there are four versions of the CP/M boot program, *CPM+.SYS*, generally available for the C128. These can be identified by the dates displayed when the CP/M system first boots up: 1 AUG 85, 6 DEC 85, 8 DEC 85 and 28 MAY 87. The 1 AUG 85 version did not support either the RS-232 port or the 1700/1750 RAM expander. Support for these were added with the 6 DEC 85 version. A minor bug in the printer routine was corrected with the 8 DEC 85 release; and full support was added for the 1581 drive with the 28 MAY 87 version. The patching points used in this article for the 6 DEC and 8 DEC versions are identical. Therefore, they will both be referred to as the "DEC 85" version.

There are also numerous modified, "unofficial" and beta test versions in limited circulation. This article deals only with the four official versions, although the experienced programmer can easily adapt the techniques explained herein to other versions.

Although the procedure is fairly straightforward, I will assume that the reader has a certain degree of understanding and familiarity with the CP/M environment. Specifically, extensive use is made of the CP/M debugging utility *SID.COM*. (Of note to long time C64 users is that this program has nothing to do with the sound chip of the same name. In this case SID is short for "Symbolic Instruction Debugger".)

Note: The addresses given in this article for the patch points refer to those obtained using SID or an equivalent debugger which loads the file being patched into the normal transient program area (TPA) memory space. Consequently, all addresses are calculated based on the beginning of the file being at address $100 (i.e. the normal start of the TPA). If you are using a hex file editor which calculates its addresses based on the beginning of the file being address 0 (such as *EDFILE.COM*), you must subtract $100 from the addresses given here in all cases to get the patch point for these other programs. For example, address $440 with SID, is actually offset $340 when you count it from the beginning of the file using EDFILE. ($340 from the start of the TPA at $100 = $440)

Starting off

The first step in customizing your CP/M system is to boot up CP/M and enter the SID environment. This is done by first putting the CP/M boot disk in disk drive 8 and turning on the computer or pressing the reset button. Once the CP/M command prompt (normally **A>**) appears, you can proceed to the next step. With a copy of *SID.COM* on the same disk as your *CPM+.SYS* file, type in:

```
SID CPM+.SYS   <return>
```

Note: *Use a backup work disk. Do not do this with your original system disk because it will make permanent changes to the operating system.*

After a few moments, the screen should display something like:

```
CP/M 3 SID - Version x.x
NEXT  MSZE   PC    END
zzzz  zzzz  0100   CEFF
#
```

where **zzzz** is a hexadecimal number indicating the length of the *CPM+.SYS* file. For the 1 AUG 85 version, it will have a value of 5D00; a value of 6400 for the DEC 85 versions; and a value of 6300 for the 28 MAY 87 version. If one of these numbers does not show up, then proceed with caution because you may not be working with an 'official' release and some or all of the patch points may be different. In any case, mark this number down because it will be needed at the end when you save your changes. The # is SID's normal input prompt: it is now awaiting your further instructions.

Throughout the remainder of this article, I will be referring to three main SID commands. These are **d** for display memory; **s** for set memory (i.e. change memory bytes); and **w** for write memory to file. The syntax for each of these commands is summarized below. Note that there is no space between the command letter and the first argument, but a space (or comma, as applicable) is required between arguments.

```
d<start address>,<end address>
s<start address>
w<file spec>,<start address>,<end address>
```

In all cases, *<start address>* and *<end address>* are expressed as hexadecimal values and both addresses are optional. If *<start address>* is omitted with the **d** command, the display starts at the current program counter value (default start at 0100, and with subsequent uses, it resumes from where it last left off), while if *<end address>* is omitted, the next 192 bytes will be displayed (12 lines of 16 bytes). If *<start address>* and *<end address>* are omitted with the **w** command, the values displayed in SID's sign-on message under **PC** and **NEXT** are used as the start address and end address respectively. For safety's sake, it is a good idea to specify the addresses explicitly when modifying bytes with the **s** command or writing the modified file with **w**. That way you do not have to keep track of the default values from the program counter. With SID, the parameters can be separated by either spaces or commas.

Screen colours

As part of the start-up routine, C128 CP/M specifies the screen colours for the characters, background and 40-column border area. Now, you must admit that not everyone will like the purple characters on a black background with a brown border that the wise folks at Commodore chose as the defaults. Our first task is to change them into something a bit more palatable.

On examining the CP/M source code, one finds that the start-up routine specifies logical colour 0 for the background, logical colour 4 for the foreground (i.e. the characters) and logical colour 9 for the 40-column border area. Armed with this knowledge, there are two ways to change the defaults: change the logical to physical colour translation table or change the logical colour codes in the start-up routine. You need only do one of the following changes, so take your pick.

The first method is perhaps the easiest, and is a good introduction to the workings of SID. In all official versions of C128 CP/M, the logical to physical colour translation table is in the same spot. Type in the following at the SID prompt #:

```
d5f0
```

followed by the Return key. (From this point on, whenever I say to type in something, you should always press the Return key afterwards.)

You should see a display similar to:

```
05F0: 00 11 22 33 44 55 66 77 88 99
      AA BB CC DD EE FF ..''3DUfw........
```

followed by a number of other lines of similar format. This first line is the logical to physical colour translation table. (If you have previously redefined your colours using the KEYFIG.COM utility, then all of the numbers may not appear exactly as shown.) Memory location 05f0 contains the physical colours corresponding to logical colour 0 for the 80 and 40 column displays (they can be different, if you so desire), while 05ff contains the physical colours for logical colour f. In each byte, the low nybble, -x, contains the physical colour for the 80-column screen, while the high nybble, x-, has the colour for the 40-column screen. Initially, all logical colours are defaulted to the corresponding physical colour numbers, hence the 00 11 22 33 etc.

Recall that logical colour 0 is specified for the background. To change this, type in:

```
s5f0
```

SID will respond with:

```
05F0  00 __
```

In this case, 00 is the current value for memory location 5f0 and __ is the location of the cursor. Type in your new hex value, say 16. (This will give a black background on the 40-column screen and a blue one on the 80-column screen.) The physical colour nybbles can be obtained from the following table (note the differences between 40 and 80-column modes):

Nybble Value	Colour	
0	Black	
1	White	
2	Red	
3	Cyan	(lt. cyan 80 col)
4	Purple	(lt. purp 80 col)
5	Green	
6	Blue	
7	Yellow	(lt. yellow 80 col)
8	Orange	(dk. purp 80 col)
9	Brown	(dk. yellow 80 col)
a	Lt. Red	
b	Dk. Gray	(dk. cyan 80 col)
c	Md. Gray	
d	Lt. Green	
e	Lt. Blue	
f	Lt. Gray	

SID will respond with:

```
05F1  11
```

...which is the next memory location to patch. We will not make any changes here for now, so press Return a few times until SID says:

```
05F4  44
```

This is the foreground or character colour location. Type in the new value, say 11, for white characters or 55 for green (like a green screen monitor). Press Return a few more times until you get:

```
05F9  99
```

This is the final colour location to patch. It can be ignored if you use an 80-column display because the border colour is only used in 40-column mode. It is generally most appealing to set this one to either the same colour as the background or a different shade of the same colour (such as light blue with dark blue, light red with dark red, etc.)

After you have completed the changes, type in a period ('.') to exit the s command and return to the main SID input prompt (#).

(Note that it is also possible to patch this colour table using the C128 keyboard redefinition utility KEYFIG.COM. Bear in mind that the logical colours to be changed to set the default screen colours are the same as those outlined above.)

The second patch method for changing the screen colours may seem a bit more complex to some because the precise location to modify depends upon the version of CP/M that you are modifying. However the technique is virtually identical to that outlined above. The colours are actually set just before the sign-on message by using a series of escape codes printed to the screen according to the following extract from the source code:

```
call prt$msg   ;call routine to print signon message

db ''Z''-''\'' ;control-Z to clear screen
(note this screen clear is not really required)

db esc,esc,esc ;prefix for setting color
db purple+$50  ;logical color 4 for foreground

db esc,esc,esc
db black+$60   ;logical color 0 for background

db esc,esc,esc
db brown+$70   ;logical color 9 for border

db ''Z''-''\'' ;clear screen
(this is the one which is required)

db ''CP/M 3.0 On the Commodore 128'' ;signon msg

db date
db cr,lf
db ''                    '' ; filler
db 0                       ; end of message

(program resumes execution here)
```

Note that the three colours need not be specified in any given order. They may also be specified as physical colours rather than logical ones. The patch locations are as follows:

Color Code	Location		
	1 AUG 85	DEC 85	28 MAY 87
Foreground	28d7	2ffb	2e01
Background	28db	2fff	2e05
40 col border	28df	2f03	2e09

All three locations are interchangeable because the colour source which is actually set by a given location depends only on the value of byte. To set a given location as a logical foreground colour, add $50 to the colour table numbers given above. To specify the location as a logical background colour add $60, and for the 40-column border colour add $70. For example, if you wish to specify the foreground as logical colour 1, the background as logical colour 7 and the border as logical colour 3, the byte values would be $51, $67 and $73.

If you prefer to specify the colours as physical colours (i.e. the logical to physical colour translation table is bypassed and the specified value is used directly as the colour), the corresponding adders are $20 for the foreground, $30 for the background and $40 for the border. For example, if you used values of $21, $32, $46 you would always have white characters on a red background with a blue 40-column border, regardless of how you had defined your logical colours.

Use SID's s command to set the appropriate locations to your desired new values. (Don't forget to type in a period when you are done to return to the main SID prompt.) It may appear that the locations given for the DEC 85 version are out of sequence. I assure you, however, that they are in correct sequence. The illusion is created by the obscure method in which most of the *CPM+.SYS* file is stored on disk. It is stored in 128-byte records in reverse order (i.e. the first record after the file header is placed into the high end of the computer's memory and proceeds downwards). This creates apparent discontinuities in parts of the file which happen to cross over one of the 128-byte record boundaries. As we shall see in the next example, this can also create some minor confusion when trying to patch across a split record.

The sign-on message

As listed above in the excerpt from the CP/M source code, a sign-on message is included for displaying on the screen when CP/M first boots up. In its default form, this is a very boring message consisting of "CP/M 3.0 ..." etc. followed by the date. Being in the mood for customization, we can change this to anything that we like, up to about 50 bytes total length.

Wouldn't it be more interesting for your computer to display a personal greeting each time you started it up? How about "Good morning Fred, this is HAL speaking"? Or "Don't bother me now, I'm thinking"?

The patching procedures for the DEC 85 and 28 MAY 87 versions are simple and straight forward. With the DEC 85 version, you have 53 bytes to play with at addresses 2f05 to 2f39. (Not including an initial CTRL-Z ($1a) at $2f04, to clear the screen before printing the message.) With the MAY 87 version, you have 54 bytes from 2e0b to 2e40, with the clear-screen at $2e0a.

Your custom message can be created by using a variation of SID's **s** command. Type in s<*start address*> where start address is the previously mentioned value for the CP/M version you are using. SID will respond with something like:

```
2F05 43
```

and will await your input. Instead of typing in a single hex value as before, you can type in your desired ASCII string, preceded by a quote:

```
"Fred's computer. Good morning...
```

and followed by a Return. *Note that you do not use a trailing quote*. Anything entered after the first quote is interpreted as part of the string. Using a trailing quote will cause this quote mark to be included in your message.

In addition, you should not use trailing spaces unless you want them to be included in your message. SID will display the next available memory location after your change followed by the current value of this location and the input cursor. Type in a period and Return if you are done or more text if you are not.

You can include linefeeds (hex $0a), Returns (hex $0d), as well as other cursor and screen control escape codes in your message. These are easiest to enter with the normal **s** command outlined above for changing the hex value of individual bytes. (When doing this, you should use the **d** command frequently to keep track of where you stand.)

When you have finished your custom message, check your overall work with the **d** command to see how much padding you should add:

```
d2f05    (or d2e0b for the MAY version)
```

Your message should be padded out with a series of blanks ($20 byte values) until you reach $2f3a (DEC version) or $2e41 (MAY version), at which point you should have a hex 0 byte. This 0 byte is very important as it serves as an end of message marker. The print message routine that was called at the start of the sign-on will resume at the address immediately following this byte. Misplacement of the zero byte terminator may cause a system crash.

Unfortunately, the sign-on message for the AUG 85 version is split across a record boundary. The first half of the message consists of 31 bytes located at $28e1 to $28ff and the second half is 24 bytes long at $2800 to $2817. The initial clear-screen is at $28e0 and the 0 byte message terminator is at $2818.

It is especially important in this case to keep track of your byte count when entering the new message becuse if you overshoot the available space, you will corrupt your system disk.

The RUN/STOP key

Most people who are familiar with the operation of the C128 in native mode (and for that matter, most other Commodore 8-bit computers) use the RUN/STOP key as sort of a 'soft' reset button to halt the execution of a BASIC, or even a machine language program. However, you may have also discovered that this does not work in CP/M mode. (How often have you been in CP/M mode trying to abort a program and pressed RUN/STOP out of habit?) The equivalent general program exit command and soft reset in CP/M is CTRL-C. (That is, hold down the Control key and press the letter C key at the same time.)

This next patch modifies the keyboard decoding tables to assign a value of CTRL-C (hex $03) to the RUN/STOP key. The patch point is the same for all three versions of C128 CP/M: $058c. Change the byte at this location from 0 (representing the equivalent of no action) to 3 (representing a CTRL-C). That's all there is to this patch. Now when you press RUN/STOP in CP/M mode, it will be the same as pressing CTRL-C to exit a program.

The RAM disk

The AUG 85 CP/M version does not support the RAM disk, so it will not be discussed further for this topic.

A) The disk label: CP/M has a convenient method of assigning a name (or volume label) to a given disk to help you keep track of which disk is which. (This is similar to the convention of naming a Commodore DOS disk during formatting, except that it can be done at any time.) The name is assigned using the CP/M *SET.COM* utility.

The label is recorded as a special entry in the disk directory which is normally invisible. (You cannot see the volume label when you do a DIR command for the disk directory.) Certain operating system extensions make use of the directory label as a way of telling which disk is currently in your drive. It can also be a simple method of 'personalizing' your disks.

So far so good, or at least one would think so. But - and here comes the cruncher - you should be wary of assigning a name to your RAM disk (drive M:). The reason for this is quite simple: it already has a name which is also used as a flag to control formatting of the RAM disk on a system boot!

When CP/M boots up, it checks for the presence of the RAM disk by looking for the RAM expansion controller (REC) registers. If it finds the REC to be present, it then checks the first entry of the RAM disk directory for a 'key' to see if the RAM expander has been initialized as a CP/M RAM disk. This key is the disk label "ERTWINE VON". (Von Ertwine was the chap responsible for adapting CP/M to run on the C128.) If this label is not found, the boot process will 'format' the RAM disk by erasing the directory area with hex $e5's then writing this label to the first entry, thus losing any data which may be present already.

There are a number of reasons why you may want to preserve data when switching modes or rebooting your system. The most obvious is to recover from a system crash. If you had created or edited files on the RAM disk without saving them to a floppy and then subsequently had a crash or lock-up, you may want to be able to recover the files when you reboot. Normally, everything in the RAM disk would be preserved, providing you did a reboot by pressing the reset button momentarily. However, if you rename your RAM disk using SET, the key will no longer be present and your data will not be preserved when going from CP/M to C128 mode and then back again.

The DEC 85 version is relatively simple to patch. The MAY 87 version is a bit more difficult, again due to the patch area being split across a record boundary.

For the DEC 85 version, the label is located at $1e5a to 1e64. To see it type in:

```
d1e59
```

The text "ERTWINE VON" should be shown on the first line of the dump. The first character is a hex $20 (ASCII space) which indicates that the entry is a directory label. The last byte is a 01. Neither of these should be changed. Use SID's **s** command to change the text of the label:

```
s1e5a
```

SID should respond with:

```
1E5A 45
```

To make the changes, type in your new disk label as a text entry preceded by a quote such as:

```
"MIKES DISK
```

You must include the quote mark at the beginning. You have eleven characters to play with and they should be in the form of a legal CP/M filename (i.e. all uppercase with no reserved or special symbols such as ? or *). Unused locations should be padded with spaces. When you have pressed Return, SID will display the next memory area, which might be:

```
1E65 01
```

Type in a period (.) followed by Return to signify that you are finished. You can check your handiwork by typing in **d1e59** again.

Now for the MAY 87 version. The first three characters of the label (ERT) are at $1cfd to 1cff, while the remainder (WINE VON) are at $1c00 to 1c07. To see this, type in:

```
d1c00 1cff
```

To change the label, you will have to change the memory bytes in both locations, bearing in mind the number of bytes used at each location. This can be done in two steps:

First, use **s1cfd** to change the first three bytes in the label at $1cfd to 1cff. Remember to type in a period when you have changed these three bytes to return to the SID command prompt.

Second, use **s1c00** to change the remainder of the bytes at $1c00 to 1c07.

B) The drive code: C128 CP/M automatically assigns the RAM disk to drive M:. While this is good for most applications, there are a few CP/M programs (mostly those designed to work under much older versions of CP/M) which will not accept anything over D: (such as M:) as a legal drive specifier. In this case, it is wise to change the RAM disk assignment to some other letter, such as B:, C:, or D: (assuming that you do not have a disk drive already so assigned). Changing the drive assignment invloves making a few patches to the DRIVETABLE and optionally to the code which checks for and initializes the RAM disk. First the DRIVETABLE.

For both the DEC 85 and MAY 87 versions, the DRIVETABLE is located starting at $651. It contains a set of 16-bit vectors, one for each drive letter, to the disk parameter block for each of the drives. (If no drive is assigned to a given letter, the vector has a value of 0 0.) The first thing we must do is to "de-allocate" drive M:. This can be done by setting both locations $669 and 66a to 0. The second step is to allocate another drive letter to the RAM disk. The vector for the RAM disk DPB is $fb96. Translated to low byte/high byte format, this becomes $96 and $fb. The correct addresses to patch depend on the desired drive code according to the following table:

```
--------------------------------
Drive code   location for
             --------------
                 96       fb
--------------------------------
  A: *          651      652
  B:            653      654
```

C:	655	656
D:	657	658
E: *	659	65a
F:	65b	65c
G:	65d	65e
H:	65f	660
I:	661	662
J:	663	664
K:	665	666
L:	667	668
M:	669	66a
N:	66b	66c
O:	66d	66e
P:	66f	670

* **Note:** you should avoid assigning drives A: and E: to the RAM disk because this may cause problems with other CP/M system functions.

When CP/M boots up it checks for the presence of the RAM disk. If it is not found, the corresponding vector in the DRIVETABLE is removed and replaced with 0 0. This process assumes that the RAM disk is assigned to drive M:. Since we have just changed this assignment, it is desirable to change the vector address which will be updated. Note that this patch is not essential, but will ensure that you will not be able to access the RAM disk drive code if you do not have a RAM disk installed. With the DEC 85 version the patch address is $1e10; and with the MAY 87 version, it is $1cb3. In both cases, the contents of this byte will be $e9 representing the pointer into the DRIVETABLE for drive M:. This byte should be changed to a value from the following table which corresponds to the drive letter installed above:

Drive code	value	Drive code	value
A:	d1	I:	e1
B:	d3	J:	e3
C:	d5	K:	e5
D:	d7	L:	e7
E:	d9	M:	e9
F:	db	N:	eb
G:	dd	O:	ed
H:	df	P:	ef

The default printer

When CP/M boots up, the CP/M logical LIST device (i.e. the printer) is assigned to the physical device PRT1 (i.e. serial port printer with device #4). You may wish to use another printer device, such as device 5, or even an RS-232 port device as the default printer. (I have two separate monitors hooked to my system: one for the 40-column screen, and the other for the 80. In some cases, I use the 40-column screen as a temporary 'printer'.) The patch address to change for the default printer assignment is $28c4 for the AUG 85 version, $2fe8 for the DEC 85 version and $2eee for the MAY 87 version. Normally, this byte will have a value of $10 which corresponds to the value of PRT1 from the following table:

Printer device	Byte value
80COL (screen)	$40
40COL (screen)	$20
PRT1 (device 4)	$10
PRT2 (device 5)	$08
RS-232	$02
(DEC 85 and MAY 87 versions only)	

Multiple device assignments are also possible. For example, a value of $18 ($10 + $08) will assign the printer to both devices 4 and 5. In the case of the RS-232 port, some fiddling with the serial protocol of the printer may be required to match the default baud rate and communication protocol of the C128's RS-232 port.

The drive search chain

When CP/M is looking for a program, it can search up to four separate drives before it gives up its search and reports the equivalent of a "file not found" error. This sequence is called the "drive search chain". In the DEC 85 and MAY 87 versions, the drive search chain parameters are located at $1268 to 126b, while for the AUG 85 version, the search chain is at $0e68 to 0e6b. In all cases, the default version of CP/M contains the chain: 00 ff ff ff which corresponds to searching the currently logged drive only.

To set the search chain, the following byte values are used:

```
00 = default or currently logged drive
01 = drive A:
02 = drive B:
(... etc)
10 = drive P:
ff = filler
```

For example, if the search chain was set to:

```
00 0d 01 02
```

and you typed in a transient command such as PIP, CP/M would search the default drive for the corresponding file (*PIP.COM*). If it was not found on this drive, CP/M would then try drive M:. If it was still not found, drive A: would be tried next, then drive B:. If it was still not found after the complete search, CP/M would report back with a file not found error.

The default drive and user area

After CP/M boots up, control is returned to the user via the Console Command Processor (*CCP.COM*) which waits for your command. The CCP prompt takes the form of "{*user number*}{*drive letter*}>" (such as **3M>**), where *user number* and *drive letter* represent the "currently logged" user area (3 in this case) and disk drive (M: in this case). When you first boot up, this is normally set to user area 0 on drive A:, giving the familiar **A>** prompt. In some cases, such as when you use one disk drive to boot from, but store most of your programs on a different drive, you may wish to change this default setting to avoid having to change the drive assignment explicitly each time you boot up.

The default drive on a cold boot is controlled by the byte at $122f for the DEC 85 and MAY 87 versions and at $0e2f for the AUG 85 version. The value of this byte is 0 for drive A:, 1 for drive B:, etc., up to f for drive P:. (This byte, which ends up at offset $13 of the system control block (SCB), is also used by the CCP during the warm boot routine for establishing the default drive after exiting from a transient program. The value at this location is updated each time you explicitly set the drive from the CCP by issuing a **<*drive letter*>: <Return>** command. This patch only sets the initial value used after a cold boot.)

The default user area on a cold boot is controlled by the byte at $1230 for the DEC 85 and MAY 87 versions and at $0e30 for the AUG 85 version. The value of this byte ranges from 0 for user area 0 to f for user area 15. It ends up at offset $14 of the SCB and is also used by the CCP during a warm boot.

Extended 1581 support

In Volume 8, Issue 03 of *Transactor* (November 1987), I presented a patch for the AUG 85 and DEC 85 versions of C128 CP/M that would allow full use of the capacity of the 1581 drive. As I mentioned at the time, my 1581 disk format would not be compatible with the "official" Commodore version (i.e. the MAY 87 CP/M release). Since some of you may have obtained the MAY 87 CP/M release since making my initial patch, you may be wondering how to access the 1581 disks made with my version. Fear not, the next patch allows the MAY 87 version to read and write these early 1581 disks, in addition to the "official" 1581 disks. (The patch points are also repeated for the AUG 85 and DEC 85 versions for the benefit of those who missed them the first time around.) Change the listed bytes to the "new byte value" to complete the patch. This patch fiddles with the disk parameter table entries for the EPSON QX-10 disk type (10 x 512 sectors), so you will lose compatibility with this type, but gain an 800K disk instead.

ADDRESS			New Byte Value
AUG 85	DEC 85	MAY 87	
1404	2165	1f08	00
1405	2166	1f09	86
1406	2167	1f0a	01
1412	2173	1f16	"MG 1581 (enter as text string)

Correcting bugs

This final patch cures a bug in the AUG 85 version which prevents you from executing custom 8502 machine language routines from within CP/M. (Yes Virginia, you can switch the 8502 on from within CP/M mode and execute 8502 machine language programs!) The error is at $5cab which ends up in the BIOS 8502 portion of the CP/M operating system. Change this byte from a $c3 to a $6c and you are off to the races with BIOS function 30, group 4, subfunction 9 "User call to 8502 Code Routine" (described on page 700 and 701 of the *C128 Programmers Reference Guide*). The 8502 code at this location should be JMP (FD05), but the $6c for the JMP instruction was somehow scrambled into a $c3 by the cross assembler used to create the routine.

Closing up

Now that you have completed all of the patch work on your system disk, the last thing to do is to save a copy of it. This is done with SID's **w** command:

```
wcpm+.sys,100,zzzz
```

where **zzzz** is the address that you copied down from SID's sign-on screen at the beginning of the process. To refresh your memory, it should be 5d00 for the AUG 85 version, 6400 for the DEC 85 version or 6300 for the MAY 87 version. After SID has rewritten the file, it should display a message similar to:

```
yyyyh record(s) written
```

where **yyyy** has a value of 00B8 for the AUG 85 version, 00C6 for the DEC 85 version, or 00C4 for the MAY 87 version. After you get this message, you can exit SID with a CTRL-C.

To see the effect of your changes, you must do a cold system reboot (i.e. with the modified *CPM+.SYS* disk in drive A:, press **CTRL-<Enter>** or the reset button). When CP/M comes back on, your changes should be in force.

(If by some chance you have made an error in the patches, and you cannot get CP/M to reboot or it does something unexpected, it is probably easier to start with a fresh copy of the old unmodified *CPM+.SYS* file rather than trying to fix your modified one. It is also wise to wait a short time before you copy your modified CP/M system to your boot disks to check that you have not created any 'hidden' bugs by your patching attempts.)

At first glance, this article may seem to suggest a formidable task, especially for the novice CP/M programmer. However, it is fully recognized that not everyone will want to make all of the changes mentioned above but a combination of a few of them can add some nice custom touches to your CP/M environment. In addition, there are probably many more changes that could be made in the form of patches. I leave these to other readers to figure out.

What's Really Inside The Commodore 64?

by Milton Bathurst

Book review by Jim Butterfield

Published by DataCap, 12 Trixhal, B-4545 Feneur, Belgium

Available in North America from:

Schnedler Systems
Dept. 94, 25 Eastwood Rd.
P.O. Box 5694
Asheville, NC 28813
(704) 274-4646
242 pages, $29.95 US postpaid USA

This book is a complete and relatively good disassembly, with cross-reference material, of the ROM of the Commodore 64. The title is misleading, since there are many other things inside the Commodore 64 that could be documented: I/O chips, specialized RAM areas, and ports (pin connections and levels).

Bare bones disassembly

The book contains little else apart from the ROM disassembly. The author has done his job carefully; data tables are not confused as code, and the tricky BIT entry masks are caught well. "Immediate" data, where numeric values might sometimes be confused with address fragments, are generally good.

A disassembly is not in itself extensive documentation of ROM. What would be needed to do the job better is detailed register, memory and condition code requirements for the various routines as they are entered and as they exit. For example, here's documentation of subroutine CHRGET: Before calling, a pointer (TXTPOINTER, address hex 7A/7B) needs to be pointing at the address to be scanned; no special register or flag setup is needed. When the subroutine returns, registers X and Y will not be disturbed; register A will contain the next character from the text stream; TXTPOINTER will now point at that character, flags Z, C, and N will be affected according to the nature of the character.

If you don't have details like those of the above example, you will need to do a good deal of research before using such a subroutine from your own program. The book doesn't see such detail as part of its task; don't expect to get that information here.

The book documents ROM, not RAM... yet sometimes RAM contains important code. For example, CHRGET, mentioned above, is in RAM; at startup time, it's copied from E3A2 in ROM and placed at $0073 in RAM. It's an important subroutine that is used frequently by BASIC. But if, as you look through the book, you see JSR $0073, no amount of frenzied flipping of pages will show you a subroutine at that address.

While the code is annotated - about 80 percent of the machine language instructions carry a brief note - it doesn't explain, and does not set the stage. When you look at the math subroutines, you must know in advance how floating point numbers are set up on the 64. You must know that floating point registers will be pre-loaded with the values to be handled. When you look at the code for a command such as LIST, you must know that, upon entry, the C flag will be clear if LIST is followed by a non-numeric character, and the Z flag will be set if LIST stands alone. The book won't tell you any of this; it assumes you know it in advance.

Cross reference

A useful part of the book - not readily available elsewhere - is a cross-reference of the ROM code. This is carefully done, but it's fragmented: jumps, branches, subroutine calls, vectors, "external" addresses, and zero page addresses are listed separately, each with a special prefix; and to add to the proliferation of cross-reference entries, the BASIC and Kernal ROMs each have their own set of listings.

Fragmented addresses are generally handled well, and are carried through correctly to the cross-reference area. For example,

...continued on page 71

Macro Set 1 for C64/C128

from Xytec

Review by M. Garamszeghy

Available from:

Xytec
1924 Divisadero
San Francisco, CA 94115

$29.95 US, $5.00 S&H, add $1.80 for Canada

Macro Set 1 from Xytec is a collection of some 36 ready to use assembler source code subroutines and 60 assembler macros. It is currently available for the *Merlin* and *Commodore Assembler Developer System*, while versions for other popular assembler formats (such as *PAL* or *Buddy*) are said to be forthcoming. It provides a readily available and easy to use source of assembler source code for many common tasks faced by programmers.

Macro basics

For those who are interested, an assembler macro is really just a simple way to insert a lengthy piece of standardized source code into your program by specifying a name with perhaps a few parameters. (Not all assemblers support macros, but most of the better ones do.) Each time the macro is called in your assembler source code, a full copy of the represented code is inserted into the object code.

For example, if you wanted to open a disk file, you normally have to go through a series of steps using the Kernal SETNAM, SETLFS (SETBANK on the C128) then OPEN. Instead of manually repeating these steps each time you wanted to open a file, you could set up a macro (let's call it DOPEN) which contained all these steps. Your assembler source code may then contain some lines which may look something like (remember, the syntax for various assemblers is different):

```
DOPEN 'filename',file#,device#,channel#
```

where **filename, file#, device#** and **channel#** are your parameters for the given file. Now, each time your assembler sees this line, it takes the source code (which may be several hundred lines long) associated with the macro DOPEN, inserts the parameters at the appropriate spot, then inserts the code into your main program. By keeping sets of often used macros in standard reference libraries, you can greatly reduce the effort in writing new assembler programs.

The libraries

Macro Set 1 does just that for you. Macros and subroutines are provided for most housekeeping functions such as screen printing, keyboard input, disk file I/O, etc. A set of EQUATES for the standard Kernal entry points and other memory and system values (colour codes) is also provided in one of the libraries. (However, conspicuous by its absence is support for some of the peripherals and programming areas many people need help with such as sound, graphics, mice and RAM expansion units. In addition, although the disk label states "for the Commodore 64 and 128", there is no support for the C128 80-column VDC chip, bank switching or the C128's enhanced Kernal set.)

The macros and subroutines are divided into six libraries which can be combined with your own assembler source code. (One very nice touch is that the manual states that programs containing code developed from the Xytec routines may be freely distributed without attribution. After reading the conditions of many compilers, code libraries, etc. that basically state you cannot sell any program developed using system XYZ without the permission of Mega XYZ Corp., this is truly a welcome relief!) In the CAD-64 version, which I tested, the libraries were supplied as SEQuential data files. (Other versions of *Macro Set 1* would presumably have library files compatible with the given assembler.)

The START library contains the Kernal and system EQUATES as well as general work areas, pseudo 16-bit register handling routines and general housekeeping (i.e. screen clearing, cursor positioning, etc). The INPUT library contains a fairly sophisticated macro for keyboard input. The BSAM library has file handling macros and subroutines. ARITH contains a variety of math and number conversion routines. The libraries DYDUMP and TRACE contain debugging and monitoring type routines.

These libraries contain quite a number of tested and debugged assembler routines, some perhaps of more use to the everyday programmer than others, but all well thought out and easy to use.

START lets you use pseudo 16-bit registers for math, addressing and general purposes. It does this by setting up a series of low byte, high byte memory storage pairs which can be directly manipulated by a number of the macros and subroutines. This feature makes 16-bit math on the 8-bit 65xx processor a real snap to use.

What's up docs

The documentation (or *Programmer's Reference Guide*, as they call it) is a fairly extensive description of the available macros and subroutines grouped by library module, as well as necessary background info on how the libraries interact with each other. Also included are some of the basic concepts and assumptions used in developing the routines for each library. The manual is indexed by routine function as well as by general subject, and includes a listing of keywords and reserved words used by the various routines.

The entry for each macro or subroutine is accompanied by a short description of its function, special preparation required, which registers are affected and the syntax. Most are also accompanied by a short example of their usage. If I had one complaint about the manual, it would be that most of the examples are too brief or vague to get a really good idea of what they are trying to do.

On the down side, the supplied source code in the libraries is very sparsely commented. It seems that the degree of commenting is inversely proportional to the complexity of the routines. The easy ones are explained, while the complex ones have little or no comments to them. It is often very difficult to figure out what is going on. This is especially important if you want to modify any of the routines. (It is always nice to understand what you are trying to change, just to be sure that you are not removing something vital!)

A few detailed examples would also be nice. How about a complete sample source code program that calls a number of the macros and subroutines to demonstrate their proper usage? Something as simple as a sequential file reader would make use of a fair number of the macros and subroutines.

Parting words

All things considered, I would recommend *Macro Set 1* for all levels of assembler programmer; although it would probably be of most use to the intermediate level programmer who is competent enough to understand the basic concepts of the macro system, but not quite up to writing reams of source code. Beginners should be able to follow along with the aid of a good book on assembly programming, while advanced users may like it for the 'why re-invent the wheel' feeling that it provides them.

If a version of *Macro Set 1* is not available for your favourite assembler, an experienced programmer should be able to adapt one of the existing ones with little difficulty.

Inside the Commodore 64.... from page 69

LDA #$71, LDY #$A9 is correctly translated as LDA <TA371, LDY >TA371 ... well, almost correctly, since most assemblers would call for the "#" symbol to be retained. Indeed, that immediate symbol is lacking throughout the disassembly.

Even trickier code is handled intelligently. The IRQ vector table at $FD9B seems never to be referenced, but the book correctly deduces that it's reached with an offset from $FCBD, thus: LDA TDAF9B-8,X .. LDA TDAF9B-7,X. But the author didn't manage to unravel all the coding puzzles. For example, there's a seemingly baffling reference to $9FEA (not even in ROM!) at address AFD6. Not until you track the call to $AFA7 (from $AEEE) would you realize that the call is made with a value in A of $B4 or greater; through some odd arithmetic, this generates a value in Y of $68 or above. The $9FEA reference could then be changed to LDA $A052-$68,Y. A052 is the table of addresses for function calls, but you won't find it addressed any other way.

If you know your way around ROM code, you'll find it a handy compact reference...

Other books

You might want to consider other books in the same vein, to substitute for or supplement this information. Two other references give ROM details.

The Abacus book *The Anatomy of the Commodore 64* gives a disassembly plus discussion of programming considerations, with several examples of code. The disassembly (contained in an appendix that is larger than the book's main text) is commented, but not as thoroughly as in *What's Really Inside the Commodore 64*, and no cross-reference is supplied.

Compute! books has published *Tool Kit: Kernal* and *Tool Kit: BASIC*, both by Dan Heeb. These two books study the ROM of the Commodore 64, and that of the VIC-20, which has identical logic flow, to considerable depth. The code is extensively discussed, not just commented. Not all of the ROM code is discussed, but most of it is there.

Summary

What's Really Inside the Commodore 64 is a complete ROM listing. If you know your way around ROM code, you'll find it a handy compact reference.

Although the disassembly contains brief comments, this book would be tough sledding for learners. It assumes that the reader knows quite a bit about the 64's organization beforehand.

SFX Sound Expander

SID dethroned

Review by Richard Curcio

SFX Sound Expander and Disk, $90
5 Octave full size keyboard, $80
MIDI interface, $54
Music Maker keyboard overlay, $10 (C64 only)
FM Composer and Editor disk, $30
SFX Programmers Reference Guide, $10
(All prices in US dollars)

Fearn & Music, 519 W. Taylor #114, Santa Maria, CA, 93454. Call (800) 447-3434. In CA, call (805) 925-6682.

One of the more notable features of the Commodore 64 has been its excellent sound generating capabilities. The powerful SID chip, containing three oscillators with four waveforms each, four-part envelopes, and filter, is almost a complete synthesizer in a single integrated circuit. Since the introduction of the C64, synthesizer technology has overtaken and passed SID. For built-in sound generation, however, few personal computers have surpassed or even equalled the C64. (In fact, only two readily come to mind. One is the Amiga, with four channels of sampled sound. The other is not an Atari.) The *SFX Sound Expander* brings to the C64 a different form of sound generation: Digital FM Synthesis.

A brief explanation

The four waveforms of SID's oscillators are more or less fixed. The filter can be used to remove or emphasize harmonics or overtones of the waveform and thus simulate various musical instruments. This process is called 'subtractive synthesis'. Another method of sound generation involves the adding together of harmonically related sinewaves in varing amounts. This is referred to as 'additive synthesis'.

FM synthesis is an altogether different process. Sounds are created by having oscillators frequency modulate each other at high rates in complex ways. Being more than the gentle FM of vibrato, sidebands are generated; and by varying the speed and depth of modulation, and the interconnections between the oscillators, very complex sounds can be generated. Do all this with digital technology, and you have Digital FM Synthesis.

This same process is used in the electronic instruments made by Yamaha, and not surprisingly, the SFX Sound Expander employs a Yamaha IC.

Hardware

The *SFX* module is not a new product. It is manufactured by Commodore UK and has been available in Europe for several years. Fearn & Music is now importing *SFX* music products to the U.S. Physically, the *SFX Sound Expander* is a largish cartridge, resembling Commodore's *Magic Voice* module. It plugs into the C64 expansion port and has a trap door mechanism for the insertion of a companion MIDI cartridge, available separately for $54. The slot (and software) is *not* compatible with MIDI cartridges from Passport, Sequential, and others. On one side of the *SFX* module is a single RCA phono jack for audio output. This can be connected to a stereo or instrument amplifier. A special cable (included) allows the sounds to be played on a television by routing the signal into the C64 audio/video connector.

On the other side of the *SFX* module is a connector for an external music keyboard, also available separately for $80. This keyboard has five octaves of full-sized piano keys, is extremely light weight, and compact. It is neither velocity-sensitive nor pressure-sensitive, but its 'action' is quite good and not at all 'squishy' as electronic keyboards tend to be. Although it is an option, this external keyboard is essential if you want access the *Expander*'s more sophisticated features. The top two rows of the computer keyboard can also be used to play the *Expander*. A *Music Maker* keyboard overlay is also available at $10. This is a plastic cover with mini-sized piano keys that overhang the computer keyboard. Trying to play music without the overlay or the full-sized keyboard is extremely clumsy, tedious, and frustrating.

The *SFX Sound Expander* has nine 'voices' or sound generators, each consisting of two oscillators (called 'operators' in FM terminology), with envelope and phase generators. The software supplied with the *Expander* provides eight voices, or six voices plus five percussion sounds.

Software

The disk that comes with the *SFX Sound Expander* uses the form of copy protection that abuses the disk drive: while loading, the drive emits chattering, grinding, machine-gun sound effects. Since the software is useless without the hardware, and the hardware is infinitely more difficult (if not impossible) to duplicate, one questions the need for copy-protection. Sending a simple command to the drive, before loading software protected in this manner, can minimize the violence.

```
open15,8,15: print#15, "m-w"; chr$(106);
  chr$(0); chr$(1); chr$(133): close15
```

will cause the drive to be significantly quieter when software protected this way is loaded.

Once the SFX software is running, a menu line appears at the top of the screen, with a window in the centre displaying two music staves with bass and treble clefs. This window shows the notes as they are played. The C64 function keys are reprogrammed to provide cursor up and down, left and right, increase and decrease, or on and off as applicable to the chosen operation, plus 'Enter'. This arrangement takes getting used to, and I still find myself trying to use the normal cursor and return keys. The menu selections are SETUP, SYNTH, RHYTHM, RIFF, and DISK.

Highlighting SETUP with the cursor/function keys, a drop-down window appears with a number of options affecting the module's operation. When *Normal* is clicked on, the sound selected in the SYNTH window can be played over the full keyboard. *One Finger Chord* provides major and minor chords by pressing single keys on the lower two rows of the computer keyboard, or combinations of left-hand keys on the external keyboard. (These are actually two-finger chords, but why quibble?)

The RHYTHM selected interacts with this and *Fingered Chord* to produce auto-accompaniment. *Memory* holds a chord after the key(s) are released. Pressing the space bar cancels a held chord. The external keyboard can be SPLIT, and different sounds assigned to the upper and lower portions. The split point can be anywhere on the keyboard.

The SYNTH menu provides a selection of 12 different sounds. The disk comes with two 'voice' or sound banks. Additional sounds can be created using the companion program *FM Composer and Sound Editor*, available separately. The keyboard can be shifted +/- 1 octave. The notes displayed on the music staves don't change when the octave is shifted. The keyboard can also be transposed +6 and -5 semitones. Played notes are then displayed at their new positions. *Ensemble* halves the number of voices and assigns two voices to a key. The two voices are then slightly de-tuned to 'fatten' the sound. The RHYTHM menu provides percussion accompaniment in various styles: Pop, Rock, Bossanova, Country, etc. The drum sounds are uncannily realistic (especially the cymbals), unlike the "boom-chika-boom" of other inexpensive electronic rhythm units. It beats a metronome any day. When the percussion is active, the number of playable voices drops to six.

Summary

The overall performance of this package is excellent. The software does have a few shortcomings, however. The menu highlight bar doesn't 'roll over' from top to bottom or vice versa. When you reach one end, you have to go through all the selections to get to the other end. A rudimentary tone control is provided from the computer keyboard, but it would have been nice to have some sort of performance controls, perhaps using a joystick or paddles, to alter the sounds in real-time. And what Commodore UK calls a User's Guide is pathetic.

However, the sounds that emanate from the *SFX Sound Expander* are simply astounding in their realism. The *Strings* sound is especially convincing, particularly when *Ensemble* is enabled. With the optional *Composer/Editor* program, any sound found to be less than satisfactory can be shaped to the user's liking. A C64 owner really can put together a DX7-like instrument for a fraction of the cost. The mighty SID has been dethroned.

FM Composer and Sound Editor

The companion software for the *SFX Sound Expander* is *FM Composer and Sound Editor*. This package enhances the performance of that device, and is indispensable if you want to create your own sounds and compositions for the *SFX* module.

The Disk: The software employs the abusive form of copy protection that bangs the disk drive head around. While there may be some justification for copy protecting this disk, there can be no justification for the method used. After the initial head-banging, a screen appears with the choices *Editor* and *Composer*. After making your choice, the drive is subjected to some more abuse before the selected program appears.

FM Composer: The skimpy booklet that Commodore UK jokingly calls a "User's Guide" claims that this program is "powerful, yet easy to use". This is only half true. The *Composer* is indeed extremely powerful. It is also extremely difficult, and a user must be prepared to spend a lot of time experimenting and learning its features before attempting to create a musical score.

FM Composer allows the *SFX Sound Expander* to play back nine-part music scores polytimbrally. In other words, each part can have a different sound. With the companion MIDI cartridge, each voice can receive notes over any of 16 MIDI channels. An external MIDI-equipped device can set the tempo for playback (MIDI clock). Any voice can be turned on or off for playback, or slightly detuned from the other voices.

In its "clear memory" state, the *Composer* has memory available for over twelve thousand "events". For composition, many common and uncommon time signatures are supported, including 5/2, 7/16, and 9/4. Notes can be entered from the computer keyboard, the optional external keyboard, or via MIDI. Rests are entered with the space bar. When enough notes and rests of the proper durations have been entered to fill a measure of the time signature selected, the program automatically inserts a bar line. Key and time signatures can change within a piece of music. Triplets and dotted notes are supported. The instrument sound can change within a part.

A score entered with this program can have all the conventional music notations. Ritardando, sforzando, tenuto, and Da Capo are a few of the terms used in the instruction booklet. Obviously, a more than superficial knowledge of music terminology is needed to use this program to its full potential.

On playback, the piece can be shifted +/- 2 octaves or transposed +/- 6 semitones. Different sounds can be assigned to the nine parts from a library of 64 sounds. Additional sounds can be created with the *Editor* described below.

The list of features goes on and on, and perhaps that's the main problem with this program: it tries too hard. The so-called User's Guide doesn't provide much guidance and the single help screen isn't very helpful. Still, judging from the demo piece included on the disk, someone willing to spend the time necessary to learn this program will be able to do amazing things with the *SFX Sound Expander.*

FM Sound Editor: This program enables the user to develop new sounds for the *SFX Sound Expander.* The sounds created can be used by the *FM Composer* or the basic software included with the *Expander.* It also functions as a nine-voice polyphonic synthesizer.

The SETUP menu allows selection of one of 64 different sounds. The upper and lower portions of the external keyboard can have two different sounds. The keyboard can be split at any point. The keyboard can be shifted up or down one octave, or transposed. With the MIDI interface, the unit can both send and receive information over all 16 MIDI channels and "Omni".

A simple drum sequencer is included, with five percussion sounds. The sequence is 32 steps with no provision to alter the length or save the sequence. Drum events can be entered from the keyboard a step at a time, or in real time. Ten different drum 'kits' are provided. Again, when drums are active, the number of manually playable voices drops to six.

The *Edit Sound* screen is the real meat of this program. With it, any of the 64 sounds in the library can be tailored to the user's liking. Because of the complexity of FM programming, certain compromises are made. The screen presents several 'sliders' affecting various qualities of the sound. *Brilliance* and *Volume* are self-explanatory. Where the compromise is obvious is in the *Envelope* control. Instead of the four-part envelopes familiar to SID programmers, a single slider selects one of 255 preset envelopes. No graphic representation of the envelope is offered.

Because an FM voice consists of two oscillators, control over the two pitches is provided. A slider labelled *Expander* controls the feedback between the two oscillators. Each oscillator can have different levels of *Vibrato* and/or *Tremolo*. Once you've got the sound just the way you want it, it can be stored in the library and the whole library saved to disk. The first 12 sounds of a library can be saved as a 'Voice Bank' to be loaded by the Expander's basic software.

Fruit Machine: This is a silly name for a very clever process. Rather than assemble a sound from scratch, the screen shows a display similar to a slot machine, with little pictures of different musical instruments. Entering the *Go* command, the wheels are spun. When they stop, you can press a few keys and if you like the sound, go to the Edit screen and make adjustments, or spin again for a different sound combination.

Overall: My main complaint about this software is the method of copy protection employed and the non-integrated nature of the separate programs. If you're in the *Editor* and you want to exit to the *Composer*, the computer must be reset and the loading process, replete with drive-abuse, begun all over again.

With its many features, two shortcomings of the *Composer* are especially annoying. On playback, the screen display is static, stuck wherever it was when the Play command was given. This makes locating one's mistakes difficult. This is compounded by the fact that when you change parts, the display jumps to the start of the new part, rather than the location corresponding to the point where the previous part was exited. Furthermore, no numbering of the measures is provided. The inadequacy of the *Composer* instructions may discourage some users. This would be a shame, because it appears to be as powerful as claimed. Diligence and patience on the user's part are clearly necessary. In contrast, there's little to find fault with in the *Sound Editor.*

The *SFX Expander*, combined with the *Editor/Composer*, brings sophisticated synthesis to C64 owners at a very low price. One wonders why Commodore didn't make this product available in this country sooner.

Also available: *SFX Programmer's Reference Guide*, $10. This 31-page booklet, titled "Das Musik Geschaft", is more specifically about the Yamaha YM3526 IC than about the SFX module. The chip is accessed in a manner similar to the C128 80 column VDC. Only two locations appear in memory, one to write a register address, and one to write the data or read the chip's status. The booklet provides no programming examples in 65xx or any other ML, but does explain the numerous registers and functions.

I am in the process of dissecting the SFX module, and the following information is, at this time, tentative: The matrix for the external keyboard occupies eight locations at $DF08-$DF0F. The YM3526 lives at $DF20. Incomplete address decoding causes 'images' of the matrix and Yamaha chip to repeat throughout I/O2. The YM3526 is not clocked by the computer's clock, but instead has its own 3.5MHz crystal oscillator. There is the possibility that a variable clock can be substituted for fine tuning or pitch-bend. The connector for the MIDI cartridge appears to have some address lines swapped or shifted. This would explain the incompatibility with third party MIDI cartridges. As more is learned about this piece of hardware, I expect programmers on this side of the Atlantic will develop their own software for it, perhaps a C128 version that makes use of the larger memory, built-in windows, high resolution routines, and 80 column display.

X-10 Powerhouse Computer Interface

Control the world with your C64

Hardware review by Noel Nyman

X-10 Powerhouse Computer Interface

Manufactured by X-10 (USA) Inc
185A LeGrand Ave
Northvale NJ 07647

Available from several sources including

Computer Direct, Inc.
22292 North Pepper Road
Barrington IL 60010

800-BUY-WISE or 312-382-5058 (orders)
312-382-2882 (technical assistance)

$39.95 (US) plus shipping in Winter 1989 catalog

Required: C64 (or C128) and one 1541/1571 disk drive

When I first bought a house, I wanted to automate everything. In those days, computers were too expensive to dedicate to mundane things like turning lights on and off. So, I set up a system using logic gates, programmed from a patch panel. I ran extra wires where needed, and used three different voltage levels. I maintained the spirit, if not the letter, of the electrical codes. The hardware cost about $25 for each light switch.

With the X-10 Computer Interface and a C64 or C128, anyone can create a much more elaborate system than my original. It requires very little electrical experience, no extra wiring, and only a few minutes to install. Unlike my logic gate circuit, it's easy to program. It features an on-board clock/timer and full battery back-up.

The computer is used only to initialize the interface. Once programmed, it can be disconnected from the computer and operated in stand-alone mode.

The Computer Interface originally sold for over $150. In recent years it's been drastically discounted through mail order outlets. At $40, it's close in cost to the standard X-10 controllers.

The X-10 System

X-10 (formerly marketed as the BSR System) controls 120 VAC lights, lamps, and appliances by using power line carrier transmission. In simple terms, an interface is plugged into an electrical outlet in your house. Under program or manual control, the interface sends a coded radio signal through the 120 VAC wires to "control modules".

The modules are usually small boxes that plug into wall receptacles. The device to be controlled plugs into an outlet on the module. Another type of module is designed to replace standard wall switches for incandescent lights.

The modules designed for lighting applications can turn the lights on or off, and dim to any selected level. The same modules can be used for non-inductive appliances, things that don't have motors. Appliance modules are available that can switch any load up to 500 watts. They use small relays, so loads connected to them cannot be dimmed.

Each module has two dials or rotary switches. One dial selects letters from A to P (16 possibilities). The other dial selects numbers from one to sixteen. Each module can have one of 256 addresses composed of a letter and a number.

There are many types of interfaces available to control modules. Most have a selectable letter address range, using a switch or dial. This is called the "house code". The house code is not changed during normal operation. Depending on controller style, there are four to sixteen switches. Each controls a module, which is also set to the pre-selected house code, by number. There may be additional switches to dim modules once they are selected. Some interfaces provide switches for "all lights on", and other functions.

Most interfaces use manual switches. Some respond to remote control devices, either infra-red or radio. Special interfaces answer the phone and respond to touch tone signals. Others are designed for use with burglar alarm systems. Some have internal clocks and can be pre-programmed to execute commands based on time and day. You can have many interfaces in different locations to control the same, or different modules.

The Computer Interface and the C64

The Computer Interface has eight manual switches and a DIN connector. It comes with a special DIN to user port connector for the C64/C128, and a disk of software in 1541 format. A nine-volt battery is used as a power failure backup.

To use the Computer Interface, you just plug it into a wall outlet and the C64. As with any user port device, be sure the computer is turned off before connecting the cable. Start the support program by RUNning the first program on the disk.

The program is largely menu driven. You can use the keyboard or a joystick to select menu options. After setting the internal clock with the day of the week and time, you're asked for a house code. This code will be used with the eight manual switches to allow the interface to work as a stripped down standard controller. At the main screen, select the *install* option.

The program displays a 'menu' of rooms in a house, and front and rear views of the exterior. The names of the rooms can't be changed. So, if you want a module in the den, you'll probably choose the 'spare' room. Your family room might become the 'guest' room. Actually, you could put all the modules in one 'room' for ease of programming, regardless of their locations in the house.

In the empty room, you'll have several possible module locations shown as red squares. Some are on the floor, others on the walls and ceiling. As you point to a square it turns white. After selecting a square, you choose *lamp* or *appliance*. Appliances can't be dimmed, so the program won't give you that option if you choose *appliance*.

Next, you're given a choice of lamps or appliances to use. These are cute pictures made of multicolour sprites. There's a variety of appliances or lamp styles if you picked a location on the floor. Wall locations give you only wall lamps and thermostats. Ceilings can have hanging lamps, or the generic 'custom' appliance, a "C?". After installation, these pictures will appear at their locations whenever you select this room.

Each module is assigned a house code and number. The program will default to the next available number for the house code assigned to the Computer Interface. But, you can change this to any of the 256 combinations. One of the features of the Computer Interface over other interfaces is that it can use many house codes simultaneously.

After installing the modules throughout the house, you push **Q** to go back to the main screen and select *operate*.

Again, you move through the house to select a room and module. The program lets you set on and off times for each module. You can select several times for each device. You can also select one cycle (now, today, or tomorrow), or periodic cycles (every day or specific days). You can select full or dim percentage for lamps and wall switches. You can use exact time, or 'security' for slightly varying on/off times. You can also group several modules to use the same timed cycle.

When done programming, just turn off the computer and unplug the Computer Interface. A red LED (Light Emitting Diode) on the Computer Interface flashes slowly when the unit is disconnected from the power line.

Plug the Computer Interface into any outlet. It will now cycle the modules you've programmed. You can also use the manual switches to control modules one through eight that share the interface's house code. The Computer Interface cannot dim lamps manually.

A feature unique to the Computer Interface is the ability to save the stored module program on disk. You can keep several programs for various purposes and load them into the interface, saving joystick time. The disk also has utilities to control devices directly and change the time in the interface without using the menu program.

Advantages and disadvantages

The X-10 system system is incredibly easy to use, especially if you've ever spent hours crawling about under a house installing a wired system! Just plug things in and move the joystick. The wall switch modules do require a screwdriver to remove the wall plate and old switch. You'll also use wire nuts (supplied) to connect the module. Be sure to turn the power off at the breaker panel before installing wall switch modules.

I prefer the wall modules to conventional switches, even if they aren't controlled by interfaces. The X-10 wall modules are push-on, push-off. It's easy to turn on a light with your hands full. Just push on the switch with your elbow.

A disadvantage is a small mechanical slide just below the push button. It turns off the light when pushed to the left. X-10 says this is a safety feature to disconnect power when replacing lamps. It's easy to hit that slide switch instead of the main button. If the room is dark, you may try several presses before discovering, by touch, that you've inadvertently moved the slide.

You cannot dim lights manually at the switch. It can only be done with an interface. You can overide the interface brightness level by turning the controlled lamp on or off. Lights to be dimmed are first turned on at full intensity, then brought down to the desired level.

The lamp modules that plug into outlets have some extra intelligence that was a pleasant surprise. If you have a lamp plugged into a module, you can turn it on by throwing its normal socket switch or pull chain. It may take two or three tries, but the module will sense the switch activity and supply power to the lamp. If you turn off the lamp using the 'normal' switch, the module won't be able to turn it back on. The X-10's are good, but not that good!

There are specialized modules for 240 VAC appliances, such as water heaters. There's a thermostat module that mounts below your normal thermostat. It turns the furnace off by heating a small coil to trick your house thermostat. It's a bulky arrangement that you may find aesthetically unpleasing. Special 'two way' modules are used for two wall switches controlling the same light.

X-10 modules are sold by most hardware and department stores, some computer stores, and Radio Shack. They may appear under various brand names and have slight differences in appearance. They all work with the Computer Interface. Some of the specialty modules may only be available from X-10 (USA) directly. Check for sales and mail order sources. Modules and interfaces are often discounted.

The X-10 signals will generally travel through the power lines as far as the nearest utility transformer. That means your neighbor's interface signals may reach your house. If you are on friendly terms, just select separate house codes. If you don't like each other, X-10 module wars may break out.

The Computer Interface has worked well for me. I use it to turn on lights using the 'security' feature. We control lights that would normally be on, so the house looks 'lived in', even if we're not there. Since many modules can share the same code, it was easy to control several Christmas displays with one switch.

I did find the kitchen light on at odd times. The Computer Interface programming was correct. But, the light still came on by itself occasionally. I solved the problem by changing the module code. It's unlikely that radio noise on the power line was causing the problem. The X-10 signal is complex, and sent twice for verification.

Another feature when using security is that several on or off signals may be sent to a module. At my apartment, I used a module to control the computer room light. It was the most visible room from the street. Since I normally quit around 11PM, I set the program for a 'security off' at that time.

Of course, I was often working well past 11PM on a pesky program bug. The interface would turn the light off at 10:50 or so. I'd reach over to the wall and turn it back on. Maybe ten minutes later the light would go off again. This might happen once, twice, or not at all. If no one was in the room, several off cycles wouldn't be important.

I recommend the Computer Interface system. It's much cheaper than the hardware project I built years ago. It can be costly if you decide to control several devices. But, you can't replicate the electronics for the prices X-10 charges.

Beyond the Computer Interface

Most X-10 systems lack an 'input' to the interface from the real world. You can't turn lights on when a door is opened, because you can't detect a 'door open' switch. One solution is to leave the Computer Interface plugged into the computer. The joystick ports can be used for switch detection, and a BASIC program can operate the modules. That ties up the computer, however.

A better solution for experimenters is the new PL513 module from X-10. This power line interface can receive as well as send X-10 signals.

Steve Ciarcia, of *Byte* magazine fame, also publishes a magazine devoted to computer hardware topics called *Circuit Cellar INK*. Many of the articles in the past year have dealt with advanced X-10 control. You can subscribe or get past issues by contacting:

Circuit Cellar INK
Subscriptions
12 Depot Square
Peterborough NH 03458-9909
(203)-875-2199

NewsBRK

Geos Writer 64: Timeworks, Inc. has released *Geos Writer 64*, a GEOS-based word processing system that includes a WYSIWYG preview mode and high speed text entry, and imports and exports text and graphics. The program also features a 100,000-word, built-in spelling checker; a wide variety of special effect fonts; mail merge capability; extensive formatting control; a 'fast-draft' printing mode; headers and footers; document chaining, to print documents of unlimited length; single-keystroke command option; use of mouse, joystick or keyboard to move around the document; online help screens. *Geos Writer 64* is supported by Timeworks' technical support team at no charge to registered users. The program lists for $49.95 US. Order from: Timeworks, 444 Lake Cook Rd., Deerfield, IL 60015, USA.

Geos Writer 64: Timeworks, Inc. has released *Geos Writer 64*, a GEOS-based word processing system that includes a WYSIWYG preview mode and high speed text entry, and imports and exports text and graphics. The program also features a 100,000-word, built-in spelling checker; a wide variety of special effect fonts; mail merge capability; extensive formatting control; a 'fast-draft' printing mode; headers and footers; document chaining, to print documents of unlimited length; single-keystroke command option; use of mouse, joystick or keyboard to move around the document; online help screens. *Geos Writer 64* is supported by Timeworks' technical support team at no charge to registered users. The program lists for $49.95 US. Order from: Timeworks, 444 Lake Cook Rd., Deerfield, IL 60015, USA.

Investment strategy on the C128: Strategy Software has announced the release of *The Strategist*, a C128 market timing program for investors in stocks, bonds, mutual funds and commodities.

According to Strategy Software, the typical technical analysis program approaches the investor's real question – which strategy is best? – only indirectly. It allows the user to chart issue prices against one or several indicators so that he or she can visually pick those that seem to call the turns in the market, then use them to time trades. But, says the company, these programs fail to give a hard measure of how much a given strategy would have paid (or cost) the investor had he or she used it in the real world in the past.

Starting with a historical quote file for the issue of interest and a strategy specified by the user, *Strategist* goes through the file making realistic simulated trades to see how much the strategy would have paid in real life. Then, starting with the user's initial strategy, the program goes through the historical file over and over, varying the strategy slightly each time, until it arrives at a strategy that gives the optimal payoff.

Strategist uses a high-low trading system, enhanced by the use of persistence checks to confirm buy and sell signals and exponential averages of quote-to-quote volatility to modify the persistence standards and the buy and sell trigger sensitivities.

Strategist costs $29.95 US, which includes the main program and two support programs: one creates historical files and the other tracks week-to-week price activity (in the present) for trading signals. The package also includes a telecommunications program, a sequential file reader and several years of historical quotes for a fictional insurance company. The program runs in compiled BASIC and is not copy-protected. Strategy Software, Box 14-2403, Anchorage, Alaska 99514, USA.

Troubleshooting and Repairing the Commodore 128: TAB Books Inc. has announced the publication of *Troubleshooting and Repairing the Commodore 128* by Art Margolis.

The book includes a complete set of diagnostic programs that readers can use to test their own machines. Every one of the C128's chips is detailed in a separate chart that shows the chip logic, pinouts, and voltage 'scope readings. Margolis also describes how to take the machine apart and reassemble it safely, and provides the latest chip-changing techniques. Included are a vital-chip location guide and master schematic of the C128.

The contents also include chapters on: test points; servicing the logic gates; servicing digital registers; the PLA chip; the memory management unit; the address, data and control buses; the 8563 video controller; and the power supply.

The book has 448 pages and 290 illustrations. Cost is $24.50 Can. ($18.60 US) for the paperback edition and $36.50 Can. ($27.95 US) for the hard cover version. TAB Books, Inc., Blue Ridge Summit, PA 17294-0850, USA.

1581 toolkit: According to Software Support International, it took nearly 12 months to produce the *1581 Toolkit*. The package includes the following features for the 1581 user: fast data copier; track and sector editor; byte pattern searcher; file track and sector tracer; relocatable fast loader; fast file copier; directory editor; error scanner; fast formatter; partition creator; and a *1581 DOS Reference Guide.*. The documentation is in a three-ring binder. Software Support International, 2700 N.E. Andersen Rd., #A-1, Vancouver, WA 98661, USA.

Income tax preparation: Master Software announces the release of the 1988 version of *Tax Master*, which aids in the preparation of US federal income taxes. The program is for the C64, with either a single disk drive, a dual disk drive, or two single disk drives. A printer is optional.

Tax Master 1988 covers all the new tax laws, and guides the user through the preparation of Forms 1040 and 4562 (depreciation), and Schedules A through F. It also includes the tax tables, figures tax automatically, performs all calculations, and transfers results from one tax form to another.

Tax Master 1988 includes a built-in calculator function that can be accessed at any point in the program. The calculator results can be transferred directly to the line of the tax form being worked upon. The program retails for $32 US and includes a coupon said to be good for a substantial discount on the 1989 version. Master Software, 6 Hillery Court, Randallstown, MD 21133, USA.

GEOS 128, version 2.0: Berkeley Softworks announces version 2.0 of the C128's graphic operating environment. It includes enhanced versions of *deskTop*, *geoPaint*, *Desk Accessories* and *diskTurbo*. Some of the new features include stretching and scaling of images, constrain and measure tools and new graphic shapes for *geoPaint*; support for two drives (1541, 1571 or 1581) and RAM expansion unit, multiple file selection and colour coding of files in *deskTop*; and cut and paste options and automatic opening of the first photo album on disk for the *Disk Accessories*.

GEOS 128 2.0 also includes several new applications:

- *geoWrite Workshop 128* is a word processor that features: individual paragraph formatting; left, right, centre and full justification; headers and footers; decimal tabs; full page preview; 11 fonts in seven styles and multiple sizes; mixing of text and graphics; support for multiple columns, headlines and borders; and PostScript output to the Apple LaserWriter
- *geoSpell 128* operates in 80 columns and permits viewing of dictionaries and documents while checking spelling. It allows the creation and updating of personal dictionaries and supports global search and replace;
- *Text Grabber* lets the user import text from any Commodore word processor;
- *geoMerge* provides mail merge capability.

Until April 15, 1989, *GEOS 128 2.0* is available to registered users of *GEOS 128* for $35 US, plus $4.50 US shipping and handling (add $2.45 sales tax if in California), from Berkeley Softworks Fulfillment Center, 5334 Sterling Center Drive, Westlake Village, CA 91361, USA.

Genealogical software: Quinsept Inc. has created a line of genealogical software for the C64:

- *Lineages/Starter* lets the user store data for up to 570 people, print alphabetic lists in a variety of ways, and print descendant charts and three kinds of ancestor charts on a Commodore printer.
- *Lineages/Standard* does everything found in *Lineages/Starter* and adds cross-referencing and printing of address labels, and information sheets showing what is stored for each person.
- *Lineages/Advanced* permits the use of almost any printer and up to four disk drives. The user can customize the program, for example, to have it always print last name first, or to add an occupation. It can do searches in a number of ways. A small built-in word processor permits the addition of stories to each person's data. This version also comes with telephone and mail support.
- *Family Roots* adds many more capabilities to the *Lineages/Advanced* program, including the use of function keys for quick entry of repetitive place names or surnames; tracing of a new line of relations on the screen and sending only those you wish to the printer; making a chart showing only the father's line; and so on. This program is also available for the C128 in its native mode.
- *Tree Charts* is a supplemental program to both *Lineages* and *Family Roots* enabling the user to create and print a graphical representation of the family tree.

Quinsept Inc. can be reached at Box 216, Lexington, MA 02173, USA. Telephone (617) 641-2930. Quinsept is represented in Canada by Generation Press Inc., 172 King Henry's Boulevard, Agincourt, ON M1T 2V6. Tel. (416) 292-9845.

New C128 CP/M distributor: Poseidon Electronics of New York, NY has been appointed primary US distributor of the line of C128 CP/M software from Herne Data Systems of Toronto, ON. Poseidon is well-known for its distribution of Commodore specific CP/M software and will initially handle two Herne Data products: *JUGG'LER-128*, version 3.4, and *QDisk*, version 2.1. The former program is a disk utility for the 1571 and 1581 that allows them to read, write and format over 140 types of 5 1/4 and 3 1/2 CP/M disks. The latter is a memory-resident device driver that provides CP/M mode support for the Quick Brown Box, a battery-backed CMOS RAM cartridge that operates as a non-volatile RAM disk drive. Poseidon Electronics, 103 Waverly Place, New York, NY 10011, USA. Telephone (212) 777-9515.

Basic 8 returns! Free Spirit Software Inc. has introduced several products for the C128 and C64, including a newly-enhanced version of *Basic 8*, which adds over 50 graphic commands to BASIC 7.0. Several preprogrammed *Basic 8* applications are included: *Basic Paint*, *Write* and *Calc*. Free Spirit is also offering a *Basic 8 Toolkit* featuring a point and click operating system that lets users create custom pointer fonts, patterns and icons. The *Toolkit* also allows the user to convert *Print Shop* graphics into *Basic 8* graphics files, and provides a number of disk utilities that include make autoboot, convert icon file to brush file, scratch file, rename file, toggle drive and so on. *Basic 8* has a suggested retail price of $39.95 US; the *Toolkit* is available for $19.95.

Also available:

- *Sketchpad 128* provides free-hand drawing on a 640 x 200 screen, with numerous drawing tips and fonts. Output is compatible with *Basic 8*, *Print Shop*, *News Maker 128* and

Spectrum 128. Price is $29.95 US.

- *News Maker 128* is a desktop publishing program for newsletters, reports, signs and posters. It uses standard sequential files for 'pouring' text into user-defined columns. Full page layout, popdown menus, smooth screen scrolling, font selection, cut, paste, mirror and flip are among the options available. Cost is $29.95 US.
- *Spectrum 128* is a menu-operated paint program for the C128D that can display the 16 standard colours and an additional 128 colours through colour dithering on a 640 x 200 screen. Its features include air brush, erase, mirror, multicolour, block fill or erase, pixel editor, colour editor, fonts, slide show and others. It requires the 1351 or compatible mouse and costs $39.95.

Free Spirit has also introduced some C64 products, including four graphics programs which have been licensed from Solutions Unlimited, Inc.: *Icon Factory* is a graphics conversion utility ($34.95 US); *Screen F/X* is a slideshow creation and presentation program ($34.95 US); *Billboard Maker* takes graphics from most drawing programs and converts them to 4-ft. x 3-ft. signs ($34.95 US); and *Photo Finish* optimizes the clarity of the printed image (640 x 400 resolution – $29.95 US).

ESP Tester ($24.95 US) now includes a version that will run on the C64. The program is said to use the methods developed by Dr. J. B. Rhine and the Foundation for Research on the Nature of Man to test for clairvoyance, precognition and telepathy.

MACH (Maneuverable Armed Computer Humans) is an arcade-style, shoot'em-up game for the C64 that features "the ultimate warrior of the future" ($29.95 US). Order from: Free Spirit Software Inc., P.O. Box 128, Kutztown, PA 19530, USA. Telephone (215) 683-5609.

Guitar chord ear training: *Chord Printer* is a dictionary of guitar chord fingerings for the C64 containing 19 of the most-used chord types in popular music. Users can learn the chord formulas for each type and their sound by listening to the C64 play them as arpeggios. Other options include utilities for printing hard copies of staff paper, tablature paper, blank five-fret diagrams and root-node listings for the fourth, fifth and sixth strings. *Chord Printer* is available at a cost of $19.95 US from Computers, Etc!, Dept. GCP, 4521-A Bee Ridge Rd., Sarasota, FL 34233, USA. Telephone (813) 377-1121 or (800) 634-5546 (orders, only).

King James Bible on C64: *Landmark, the Computer Reference Bible* consists of the entire King James version of the Bible on 24 double-sided disks. The *Landmark* software provides access to the disks, with individual verse references and the words of Christ highlighted in colour. A concordance of over 3,300 of the most frequently looked for words is also provided on six double-sided disks. The program enables the user to, for example, create a personal Bible, complete with notes, comments, referencing and outlining of text.

Landmark TCRB is not copy protected and can be backed up to 1581s, IEEE drives such as the SFD 1001 or to a hard disk. Burst mode is used with 1571 and 1581 drives. The speed of any search is entirely dependent on the speed of the disk drive used. Commercial disk speedup products will work with Landmark (Load and search files on one side of a disk: stock 1541 - 6:00 min.; w/Epyx *Fast Load* - 1:38 min.).

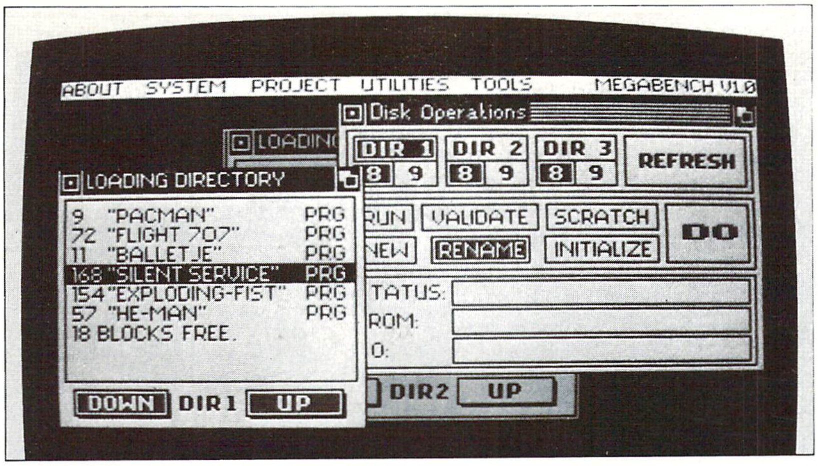
Landmark will allow you to extract information to create your own topical files. These can be manipulated with Landmark's own editing features or you can use the *File Converter* to export files that can be used with *Paperclip*, *Easy Script*, *geoWrite* (v2) or *Fleet System*. Landmark is written in 100% ML for speed. Fully menu-driven for ease of use.

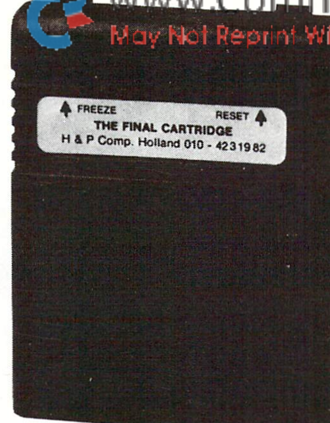
Cost is $119.95 US. PAVY Software, P.O. Box 1584, Ballwin, MO 63022, USA.

Free software from Memorex: Memorex has established a 'frequent buyer' program that lets customers build points toward free software. Titles available include the PFS series from Software Publishing Corp., and others from Accolade, Activision Disk-Count Software, Electronic Arts, Individual and Publishing International (Byte-Size). The program promotes the full line of Memorex computer supplies, including disks and paper. Instructions and a complete list of available software are on each package. Memorex Computer Supplies, 2400 Condensa St., Santa Clara, CA 95051-0996, USA. Telephone (408) 957-1000.

NEW IMPROVED

NEW IMPROVED





FINAL CARTRIDGE III®

The Best Utility Ever for Your C-64 or 128

Only! $69.95 Only!

SPECIAL VALUE: FREE Joystick and FREE 100 PARAMETER PAK,
Total Retail Value, **$39.95**, with each purchase **FINAL CARTRIDGE III**

This powerful ROM-based operating system contains easy-to-use windows and pull down menus. Allowing the user to select either mouse, joystick or keyboard, he may access over 60 new commands and functions. Let your C-64 perform like an Amiga. Various printer interfaces as well as a basic toolkit can also be accessed.

EXTENDED ML MONITOR
Does not reside in memory! Includes 1541 drive access and sprite editing. Features up and down scrolling and printer driver!

NOTEPAD/WORD PROCESSOR
Contains proportional characters and word wrap. Enables you to store and print small notes, etc.

FASTEST DISK LOADING EVER!
Contains 2 disk loaders, with speeds up to 15x faster than normal!

TRANSFORM YOUR C-64 INTO AN AMIGA LOOK-ALIKE!
Various windows such as: Preferences, Tape, Disk Windows, Directory, Printer & Clock allow you to feel as if you are working in the same friendly environment as the Amiga!

EASY-TO-USE MENU BAR
Almost any command not activated by windows can be accessed while in Basic by just typing in Box.

BASIC TOOLKIT & KEYBOARD EXTRAS
Includes: Renumbering, auto, old, delete, kill, save, 24K RAM for Basic, fast format and many, many more.

". . .I can't begin to think of a cartridge which does so many useful things. . .a tremendous value, a must item for the BASIC and machine language programmer."
—Art Hunkins, Compute's Gazette 7/87

STATE OF THE ART FREEZER
Includes variable size screen dumps (color if Epson color or NEC is used). Allows total backup of any memory-resident software on the market today! Files are packed and reloadable without the cartridge, 60K in just 15 seconds. Exits to Basic or ML monitor.

GAMES KILLER
Kills sprite to sprite and/or background collision. Can be started at any point in your games.

AUTO FIRE ENGINE
Transforms normal joystick into an auto fire!

EASY-TO-USE RESET SYSTEM
Reset your computer by the simple touch of a button!! Wow!!

"No need for all those extras when you have this C-64 assistant. . .a conventional review doesn't do the Final Cartridge justice. . .fun at a price is a rarity."
—Tim Walsh, RUN Magazine 9/87

H&P Home Personal COMPUTERS OF AMERICA
THE FINAL CARTRIDGE®

Home & Personal Computers of America
154 Valley Street
South Orange, NJ 07079
201-763-3946, dealers only, 201-763-1693

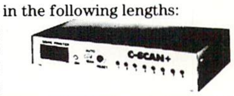
INTRODUCING SUPER CARD

Backs up any software program! Even the latest protection schemes! Plugs into your drive with only the use of a screwdriver. If anything could back up everything, this is it. 100% satisfaction guaranteed! 10 day or money back guarantee!

ONLY $39.95!

FINAL CARTRIDGE II
ONLY $24.95
Call or write for more information

Attention Schools and Educators:
C-Scan + is the ultimate network for Commodore computers, eight computers share one or two disk drives, and only one printer and software program is needed.
Simple installation, auto scanning and auto power on. Works perfectly with The Final Cartridge. 1 year warranty.
C-Scan + ... **$199.95**
Cables available in the following lengths:
9 ft. . . $13.95
12 ft. . . $15.95
18 ft. . . $17.95
36 ft. . . $19.95



EXTRAS AVAILABLE

*Final Cartridge I. $14.95
*C-1351 Mouse $32.95
Deluxe Joystick . . . . $ 8.95
Cent. printer cable* (optional) $19.95
* Limited quantities available.

Ordering Info:
Orders Only Call:
1-800-458-8682
MC/Visa accepted. Money orders (immediate shipments). Personal check (allow 2–3 weeks for check clearing). NY & NJ residents add appropriate sales tax. Add $3.00 for s/h.
Questions and info. call:
(201) 763-3946
Fax: (201) 763-1693

ANY PRODUCT PURCHASED FROM DATEL ELECTRONICS WILL NOT BE GUARANTEED BY H&P COMPUTER.

Free Spirit Software Inc.

"...excellent, efficient program that can help you save both money and downtime." **Compute!'s Gazette Dec., 1987**

1541/1571 DRIVE ALIGNMENT

1541/1571 Drive Alignment reports the alignment condition of the disk drive as you perform adjustments. On screen help is available while the program is running. Includes features for speed adjustment. Complete instruction manual on aligning both 1541 and 1571 drives. Even includes instructions on how to load alignment program when nothing else will load! Works on the C64, SX64, C128 in either 64 or 128 mode, 1541, 1571 in either 1541 or 1571 mode! Autoboots to all modes. Second drive fully supported. Program disk, calibration disk and instruction manual included.

Suggested retail price $34.95

SUPER 81 UTILITIES

Super 81 Utilities is a complete utilities package for the 1581 disk drive. Seperate version are available for C64 or C128. Among the many Super 81 Utilities features are:

- •Copy whole disks from 1541 or 1571 format to 1581 partitions.
- •Copy 1541 or 1571 files to 1581 disks
- •Backup 1581 disks or files with 1 or 2 1581's
- •Supplied on both 3½" and 5¼" diskettes so that it will load on a 1541, 1571 or 1581 drive
- •Performs numerous DOS functions such as rename a disk, rename a file, scratch or unscratch files, lock or unlock files, create auto-boot and much more!

Super 81 Utilities uses an option window to display all choices available at any given time. A full featured disk utilities system for the 1581!

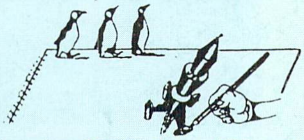
Suggested retail price $39.95

Home Designer

When the details are important!

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 printouts, manipulating little dots on a bit-map, giving up on detailed work because you can't zoom in close enough? Join the professionals!

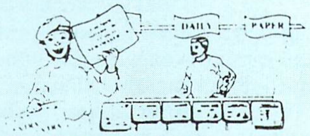
Suggested retail price $49.95



Sketchpad 128

Complete drawing system for the Commodore 128 and 1351 Mouse. Sketchpad 128 takes advantage of the crisp 80 column graphics capabilities of the C128. Smooth freehand drawing, 640 x 200 drawing screen, wide selection of dawing tips, many fonts provided. Compatible with Basic 8, Print Shop, News Maker 128 and Spectrum 128. Sketchpad 128 can be used to create 80 column artwork, slideshows, signs, posters and many other uses.

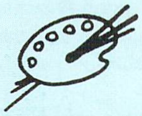
Suggested retail price $29.95



News Maker 128

Desk top publishing for the C128D (or the C128 with 64K Video Ram Upgrade). News Maker 128 can be used to create professional looking newsletters, reports, signs and posters. It can be used as a stand alone program or in combination with word processing or graphics software. It uses standard sequential files for "pouring" text into user-defined columns. Full page layout, popdown menus, smooth screen scrolling, font selection, cut, paste, mirror, flip are among the options available.

Suggested retail price $29.95



Spectrum 128

A deluxe paint program for the C128D computer (or the C128 with 64K Video RAM Upgrade). Uses 80 column display for 640 x 200 pixel resolution. Will display 128 colors! Menu operated. Requires 1351 or compatible Mouse. Features include air brush, erase, mirror, multi-color, block fill or erase, pixel editor, color editor, fonts, slide show and more. Compatible with Sketchpad 128, News Maker 128, Basic 8, 1750 REU, 1541, 1571 and 1581 disk drives. **Suggested retail price $39.95**

Basic 8

Powerful 80 column hi-res graphics programming system for the Commodore 128 or 128D computer. This popular package adds over 50 new graphic commands to standard C128 Basic. A must for C128 programmers! This new version published by Free Spirit has been upgraded and enhanced. As an added bonus several preprogrammed Basic 8 applications, such as Basic Paint, Write and Calc are included. **Suggested retail price $39.95**

See your dealer or order from: **Free Spirit Software, Inc.**
58 Noble Street
Kutztown, PA 19530
215-683-5609