THE MAGAZINE FOR COMMODORE PROGRAMMERS
JUNE 1989: VOLUME 9, ISSUE 5
Canada $4.25
USA $3.50

Transactor

- Inner GEOS - how the system fits together
- A Disk Monitor for the C128
- The 1764 Ram Expansion Unit: Add an EPROM - internally!
- Implementing a RAM disk for Abacus' Super-C
- Disk drive memory-read error exposed!
- Supernumbers III: The famous indestructible variables come to the C128
- Jim Butterfield on linked lists: a quiz program for all CBM 8-bit machines
- A Shell Sort for BASIC arrays
- Break GEOS's 31-icon barrier! - some icon programming tricks
- **Product Review**: Two Assemblers for GEOS - Berkeley's *Geoprogrammer* and Bill Sharp's *GeoCOPE*
- **Plus** Regular columns by Todd Heimarck and Joel Rubin, Programming tips in *Bits,* and more

Firebird 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**

Start Address

*What do **you** think?*

Recently *Transactor* acquired two 1581 disk drives. Consequently, authors (and would-be authors) may now elect to make submissions on 3.5" disks. The little disks (call them "flappies") are perhaps more likely to survive their journey through the postal system. *Make certain that your flappy is clearly labeled as a 1581 format disk!* Otherwise it might get swallowed up by the voracious Amigas.

Now that you know what *we've* got, we want to know what *you've* got! The other bit of *Transactor* news for this issue is the appearance of the first *Transactor Reader Survey.* This is a Commodore 'consciousness raising' exercise. The results of the survey will help to determine what you'll see in future *Transactor*s. Tell us about your system configuration, the software you use most and your likes and dislikes with regard to magazine content.

In recent years the 8-bit market has become increasingly fragmented; i.e., some users rarely leave the Power C environment, some swear by CP/M, others are committed to GEOS. Of course, there are yet others who disdain these new developments and continue to use the machines in their native environment. Such users are content to use BASIC and an assembler and thus avoid the 'overhead' of a different operating environment.

This polarization of the user community makes it difficult for any magazine to be 'all things to all people' - or even all programmers. *Transactor* has responded to these developments by attempting to provide useful information for all of these groups in every issue. Although there are topics that we haven't covered (or haven't covered recently), we feel that *Transactor* offers more support to programmers and serious users than any other magazine. But we want to know what *you* think. Participate in our Reader Survey. Don't be shy. Let's hear from you.

* * *

This issue pushes the limits with Paul Bosacki's article on the 1764 REU. You may have seen REU expansion articles from other sources but this one includes a new wrinkle: installing an EPROM. Following on the heels of Adrian Pepper's Power C RAMdisk article, Kerry Gray has us *Implementing a RAMdisk* for Super-C. Robert Rockefeller makes his first appearance in these pages with some tips on using pseudo-ops and macros with Commodore's Devpak. Anton Treuenfels *re*appears with a nifty disk monitor, among other things. Jim Butterfield discusses linked lists. Bill Coleman presents us with an overview of GEOS. Francis Kostella compares two GEOS assemblers. Richard Curcio brings robust variables to the 128, 64 and VIC. All this and so much more. Enjoy!

Malcolm D. O'Brien

Transactor
The Magazine for Commodore Programmers

Volume 9, Issue 5

Publisher
Antony Jacobson

Vice-President Operations
Jeannie Lawrence

Assistant Advertising Manager
Mike Grantham

Editors
Malcolm O'Brien
Nick Sullivan
Chris Zamara

Contributing Writers
Marte Brengle
Paul Bosacki
Bill Brier
Anthony Bryant
Joseph Buckley
Jim Butterfield
William Coleman
James Cook
Richard Curcio
Miklos Garamszeghy
Larry Gaynier
Kerry Gray
Todd Heimarck
Adam Herst
Robert Huehn
George Hug
Dennis Jarvis
Garry Kiziak
Francis Kostella
Mike Mohilo
D.J. Morriss
Noel Nyman
Adrian Pepper
Steve Punter
Robert Rockefeller
Joel Rubin
David Sanner
Anton Treuenfels
Nicholas Vrtis
W. Mat Waites

Cover Artist
Wayne Schmidt

Departments and Columns

About the cover: *Firebird* by Wayne Schmidt:

"Inspired after hearing a transcription of Stravinsky's 'The Firebird' for solo guitar (by Yamashita), itself inspired by legendary Russian folk tales, this is my *Firebird*. I am fond of the folk as well as the primitive art traditions, and the rich imagery of Russian icons and laquer painting served as models for this. This was created with *Artist 64*, modified for the 1351 mouse." - Wayne Schmidt

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.

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 :
```

LeTteRS

Responses to the ML Column: I don't know if this will help, but I generate my random numbers through the Kernal ROM. I do this by loading the accumulator with my seed value - either zero, a positive integer or a negative integer (turning on bit 7). Then I call $E09A. The random numbers will now be in registers $63-$64. (Both will fluctuate between zero and 255 quite by random; and, if not, you can always use the random number just produced as your seed which will definitely guarantee randomness.)

Here's a quick look at the code for a single random number between 0 and 7:

```
lda #1
jsr $e09a
lda $63
and #7
```

There is one drawback: it won't win any awards as far as speed is concerned.

Sean Peck, Pittsburgh, PA

Campaigning: I was struck by the idea of *Campaign* (which you described in *Transactor* 9:3) for a couple of reasons. The first was that I thought it would be fun to see it work and the second was that it asked for a solution to the problem of the generation of random numbers. I had translated into machine language an idea for a pseudo-random number generator that I'd seen in *Byte* in March 1987. It had seemed a nice exercise for writing multi-byte division and multiplication routines.

Though it was better than the random number generator with Commodore BASIC and was written to be accessed with the USR command, it seemed of limited appeal till your article came along. Having a source of random numbers, I carried out your program idea and thought I would send it to you.

The request for help in your article implied that an ideal solution would be a single memory address where random numbers could be grabbed quickly. The pseudo-random number generator that I used is complicated enough that the program takes a perceptible amount of time to calculate each point but nonetheless it moves along at a pretty good clip.

Frank van Deventer, Grosse Pointe Farms, MI

Frank's Follow-up: I subsequently ran across an article in *Transactor*, Volume 7, Issue 6 (page 27) that described 'linear maximal length shift register sequences' as a way to generate a long series of pseudo-random numbers. The method is to add pairs of numbers in a series, called a register, take the remainder after dividing by a base and replace one of the numbers in the register with this new value which is also used as the random number output. A new pair is taken each time this is repeated. This proves to be a considerably faster method. If you look the original article up, you'll see that the author gave an example program but I found that the idea could be carried out much faster by using the index registers instead of moving data and could be worked in either base 16 (4 bits) or base 256 (1 byte) to output random bytes directly.

The article indicates that the length of the series is equal to the base raised to the power of the number of items in the register. A base larger than the size of the register does not give a maximum length series but it seems to produce adequately random numbers. The real question is whether there is any bias in the series and that isn't so clear. A little inspection shows that the maximum series of odd numbers is equal to the length of the register and the maximum series of even numbers is equal to the distance between the positions of the two numbers that are added; the upper and lower taps when two taps are used. The program runs faster the larger the register and for those reasons I picked the largest register that had an optimum tap near the bottom.

I rewrote the program that I sent you earlier to use the new random number generator. You'll see that it's much faster and, at least to visual inspection, seems to work in a random fashion. It's also fast enough now to fill the screen with random bytes in an acceptable length of time. There are two versions on the disk, both of which work from BASIC. *Campaign64+* works in base 256 and *Campaign+* works in base 16. This requires only a minor change in the program. Although the latter program is marginally faster, it isn't evident to watch it. I also included the pseudo-random generator separately as rangen64+ along with a demo+ that shows access from BASIC using the USR function if you'd like to take a look at it. You'll find I think that *Campaign* is even more fun to watch when it's speeded up.

Frank van Deventer, Grosse Pointe Farms, MI

SID Sampling Rate: *The ML Column* in *Transactor* 9:3 ended with a request for a way to generate random numbers with the C64. I think the following may help.

I have found that the SID chip's noise generator can generate what appear to be truly random numbers, but there's a catch. The rate at which these numbers are produced is determined by voice three's frequency setting stored in $D40E and $D40F. At the highest possible frequency, the chip will produce about 7000 random numbers per second. This translates to a new number every 140 clock cycles. Consequently, a fast-running machine language routine can read the same number many times between changes.

One solution to this problem is to wait for the number to change before using it, like so:

```
random  = $d41b;sid random number

getrand lda random;get number
        cmp prev;"changed ?
        beq getrand;no, wait
        sta prev;for next time
        rts

prev    .byte 0;prev # storage
```

This subroutine compares the current value of **random** with what was there during the previous call and waits for the number to change before accepting it. The new number is then stored for the next call before returning the number in the accumulator. This will slow down a fast-running caller to match the rate at which the SID can produce new numbers. Note that the routine assumes that the SID has been initialized to produce random numbers.

This routine still has problems, however. A true eight-bit random number generator has a one in 256 chance of producing the same number twice in succession, but the nature of this routine eliminates this chance. Note that this is only a problem if you are using all eight bits of the number. For example, in your Campaign routine one of eight neighbors must be randomly selected. Three of the eight random bits can be used to do this. These three bits can come up the same while the other five vary, so the errors are reduced.

It seems like the complete solution to this problem requires an independent signal that indicates a new number is present in the 'random' register. It's too bad that Commodore didn't think of this when designing the SID chip. It may be possible to use some other source, such as the VIC scan line register or a CIA timer, as a source for this signal.

Randomness is a fascinating subject. It is amazing to me that such a simple concept can be so complicated in execution. I really enjoy your column and I hope it continues as a regular feature. Good luck and keep up the good work.

Mike Graham, Hopatcong, NJ

ML Wish List: I very much enjoy your *The ML Column* in *Transactor.* It was not too much over my head, and certainly not too basic either.

In a future column, I would like to see a discussion about I/O routines; for example, reading and writing to disk with various file types, DOS commands, printing, etc. I look forward to future columns.

Barry Kutner, Yardley, PA

Random bunnies: I love your new column, *The ML Column.* (Great title too!) I like how you pick interesting subjects, or at least make dull subjects sound interesting. I was very impressed with your binary division column in Volume 9, Issue 2. It answered a couple of questions I had myself.

With regard to the column in Volume 9, Issue 3... are you kidding? You had such a brilliant head long jump into the problem, only to be symied by it not being random enough?! (Actually, I skimmed the article briefly, was impressed, saw how short the BASIC generator program was, was *very* impressed, typed it in, and was disappointed to see the screen not changing. Harumph!)

Anyway, I have a solution that I figured out by simply looking at the not-changing screen. The nature of the SID output is such that it develops those irritating diagonal lines. But each byte is different from the last in that it is shifted over slightly. Two tricks to solve this, but first the explanation:

'Random' in its pure definition means an absence of pattern, but it does not mean that local patterns may not develop (like squares, triangles, bunnies, etc.). By modifying the program to run infinitely, constantly redrawing the screen, I was able to watch out, but I didn't see any bunnies... In fact, all I saw was what looked like churning water! Obviously, the SID does not do a good job of producing random displays.

So I figured, "What kind of display are we looking for here?" and I immediately thought of a static screen. About-uniform distributrion of blue and white dots, like snow on TV. (No diagonals.)

This isn't true randomness, by the way, because it doesn't allow for recognizable shapes, such as bunnies and duckies. I call this 'snow' the random pattern, a really awful oxymoron!

Anyway, two solutions, and the major problem is the pesky diagonal lines.

1) Change line 630 in the source code to:

```
630 lpchoose  lda random: adc random: adc random:
   adc random: adc random
```

Now, the gradual difference that provided the original diagonal tendency is multiplied five-fold. It's still there, but is now much more jagged, and in the 8-bit wide display, is indiscernable. It also takes a lot longer to draw the screen, because this is executed for all 8192 bytes of the screen! Not so good.

2) The diagonal tendency is caused by putting these slightly-shifted data bytes next to each other... oh, you guessed it already. Change the choose routine!

```
550 choose = *
560 bitmap = $2000
570 ldy #0
580 lpchoose ldx #32
590 lda #<bitmap
600 sta selfmod+1
610 lda #>bitmap
620 sta selfmod+2
630 lp2 lda random
640 selfmod sta $ffff,y
650 inc selfmod+2
660 dex
670 bne lp2
680 dey
690 bne lpchoose
```

I actually conjured up this solution first, simply from looking at your fill routine. I might use the first solution for the campaign routine though, to add a dash of really-more-varied numbers to the simulation. Another method would be to use the Commodore random number generator, though that is really ugly slow in comparison. Please send me your existing campaign routine!

Kevin Moorman, Calgary, AB

Precious pages: I read with interest Jim Butterfield's review of *What's Really Inside The Commodore 64* [available in North America from Schnedler Systems, 25 Eastwood Rd., P.O. Box 5964, Asheville, NC, 28813, (704) 274-4646] (*Transactor*, Volume 9, Issue 4). I use the book often, and I agree with Mr. Butterfield's assessment of it. I was surprised, however, to find two other possible supplements were left out.

One of the most useful books in my Commodore library is *Mapping The Commodore 64* by Sheldon Leemon [Compute! Publications]. It references memory locations, rather than disassembling the ROM code. That gives a picture of the dynamics of the machine missing from the other texts listed. While the Dan Heeb books [also from Compute!] give in-depth discussions of the ROM routines, they're often too detailed. For 'quick and dirty' use of the C64 ROM code, I invariably turn to Leemon's book. In a few words, he tells me what to do with registers, and what to expect as output.

I usually tell beginners that if they only buy one book, make it *Programming The Commodore 64* by Raeto West [also from Compute!]. It's not as detailed in ROM code as any of the other references. But, West lists the commonly used routines and gives excellent examples.Actually, an assembly language programmer should have all six books available.

He or she should also consider Mr. Butterfield's own *Machine Language For The Commodore 64 And Other Commodore Computers* [Brady Books], an excellent, easily understood beginning text. Marvin DeJong's *Assembly Language Programming With The Commodore 64* is another valuable addition. It gives well annotated examples of bitmap graphics, SID, and I/O routines.

To round out the library, no programmer should be without *1541 User's Guide* by Dr. Gerald Neufeld, and *Inside Commodore DOS* by Neufeld and Richard Immers.

Noel Nyman, Seattle, WA

More on household automation: After reading my review on the *X-10 Powerhouse Interface*, you may have decided that it's too limited by the lack of real-world inputs for your application.

If you prefer not to experiment with X-10's newer modules, and you have an old VIC-20 or C64 not in active use, there's an inexpensive solution. Check for the May 1986 issue of *Radio-Electronics* magazine. In the "Computer Digest" section, Chandler Sowden published a simple hardware circuit that sends X-10 signals into the power line. It uses the user port as an output, so the VIC-20 will work as well as a C64 or C128. He also provides a BASIC program to generate the control signals.

Just add switches to the joystick ports (up to 10), or heat/light sensors to the paddle inputs and you have an easily programmed X-10 system that will respond to the real world.

Noel Nyman, Seattle, WA

Some comments and a question: Volume 9, Issue 2: Joel Rubin wrote a comparison of some commercial assemblers for the 128, and also mentioned *The Fast Assembler* by Yves Han, which appeared in the January 1986 issue of *Compute!'s Gazette*. I was interested in its mention, since I use this system exclusively. A few thoughts on this very unique assembler:

Joel mentioned that FA can't print out listings. Not so! Since BASIC is still active, an invocation of the famous line **open 4,4:cmd 4:list** will do the job. Since all opcodes and pseudo-ops are tokenized, however, this can only be done while the assembler is active. And because of this tokenization, source code isn't easily transferable between FA and other asemblers. Another point: FA operates as an extension of BASIC, not a replacement for it. It can be used to write BASIC programs that use the added features (indentation of lines is supported, as is the use of binary and hex numbers). Because you can write both BASIC and ML programs with the assembler, you have to manually enclose the ML part inside a for/next loop to implement the multiple passes required by a label-based assembler.

Volume 9, Issue 4: Jim Butterfield reviewed *What's Really Inside the Commodore 64?* by Milton Bathurst. Jim mentioned Abacus' *The Anatomy of the Commodore 64*. My advice is: forget it! The source is sparsely commented and unindexed. Without a memory map you'll go nuts trying to find the routines you want. The programmer's reference section is full of errors. Personally, I recommend Dan Heeb's *Tool Kit* books, published by *Compute!*. Regardless of what you may think about their magazines, Compute!'s programming books are generally hard to beat.

I'm working on a large-scale filing program and need some help on a software project I want to undertake. The filing program will be written entirely in BASIC and will utilize a string arry which occupies about 30K on its own. Obviously, this isn't all going to fit in BASIC workspace, and I'm toying with the idea of setting up a bank of the 1764 REU (say, bank 0) as a "phantom computer". The idea is to copy both ROMs, zero page, vectors, the stack, and the string array to their appropriate spots in expansion memory, then switch it all in whenever I need to access the array. Parameters and results could be transferred back and forth between the 4K blocks starting at $C000.

Of course, there are several considerations when doing this. The BASIC pointers in expansion RAM would need to be modified for the 'new' contents of BASIC workspace. I would have to perform the switch with an ML routine and interrupts would have to be disabled, since there will be no I/O. The question is, will this work? If it's possible to pull this off, how do I make the 6510 recognize a bank of expansion RAM and how do I select which bank it will look at? Any information you or your other readers could give me would be greatly appreciated. My address is: 16925 Morrison Ave., Southfield, MI, 48076.

You put out a great magazine! Keep up the good work!

Howard I. Goldman, Southfield, MI

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.

Super-C BIT
Kerry Gray, Salinas, CA

A Super-C program is loaded from $0801 like BASIC. Its first two bytes form a pointer to the first instruction in the program. You can write stand-alone assembler language programs that can be run from the C environment if you follow this convention. Your program should exit with a JMP $0400 to return control to the C shell: Don't use an RTS.

Your other file copier

I had a 457-block ARC file that I wanted to move from a 1581 to a 1571. My file copier wouldn't budge this monster! I came up with another idea. I booted GEOS and, sure enough, managed to copy the file. Not only will GEOS handle very large files, but you also get the added bonus of the GEOS disk turbo.

The Tasmanian Datafier!
Elaine Foster, Launceston, Tasmania

When you want to make a BASIC Loader for your own or other ML program, it is very nice directly to translate the data in RAM into BASIC DATA lines. This program does this very simply, and it is only five blocks long compared to a 29-block commercial equivalent. It does require that you supply the start and end addresses of the ML program to be translated. The end address will be at 45-46 after loading, and the start address will be the first two bytes in the first sector of the program on the disk - all in the usual low byte/high byte format. Many utilities cartridges these days also automatically give the start address and end address of a loaded program.

With the ML program in memory, this one is loaded into the BASIC workspace and run. It prompts you for the start address and end address and the starting line number for the DATA statements. By using Dynamic Keyboard methods (lines 50190-50230) it then makes the DATA lines and ends with the usual FOR/NEXT reading and poking loop (lines 50240-50250). The loader program is then ready to save to disk.

Lines 50150 and 50160 ensure that all data elements are aligned (*Transactor* style), which makes them much easier to read and to copy. This program cleverly erases itself when it has done its job, while leaving the new DATA statements intact! This means that when the new DATA lines have been made the program may be saved 'as is' or you can begin entering lines of BASIC or append another BASIC program.

The Deleter is a form of selective NEW, and the details are given in the source code shown (*deleter.src - provided and presented in Speedy Assembler format - MO*). You can see that it scans BASIC from $0801 onwards, looking at each link address and at each line number in turn. If the line number is not (in this case) 50000 it goes to the next link and next line number, and so on. When 50000 is found, it backs up two bytes, adds terminating zeros and adjusts the end address bytes of locations 45 and 46 to suit.

The start address of the Deleter routine (53000 in this instance) can be any one where there is room for 67 bytes; this is possible because lines 590-600 replace a JMP by a BEQ, a relative assignment. This is nice, because it allows you to avoid any conflict with the ML program being examined. If you wish to use the Deleter routine alone (BASIC Lines 50010-50080), it may be adjusted to delete from any desired line number: see REMs in lines 50030 and 50040.

Listing 1: *The Tasmanian Datafier!*

```
II  50000 rem -- the tasmanian datafier! --              transactor 9-5
JL  50010 rem -- ml to delete line 50000+ :
NK  50020 poke53280,3:poke53281,1:poke646,6:dima$(9)
KB  50030 print"{clr}{down}{down}{rvs}basic loader for ram data "
NF  50040 data169,001,133,251,169,008,133,252,160,000
NI  50050 data177,251,133,253,200,177,251,133,254,200
DN  50060 data177,251,201,080,208,029,200,177,251,201:rem 080 = lb for line 50000
BO  50070 data195,208,022,136,136,169,000,145,251,136:rem 195 = hb for line 50000
IG  50080 data145,251,200,200,152,024,101,251,133,045
DO  50090 data165,252,133,046,096,166,253,134,251,166
PP  50100 data254,134,252,169,000,240,197
AA  50110 c=53000:forx=ctoc+66:ready:d=d+y:rem - c anywhere where room for 67 bytes
IE  50120 pokex,y:next:ifd<>10888thenprint"error!":end
EH  50130 print"{down} (enter 0 to exit)"
MA  50140 input"{down}{down} beginning address";ba:input"{down} ending address";ea
NE  50150 ifba=0orea=0thenend
```

```
CM 50160 input"{down} first data line #";l
KF 50170 ifea-ba+l-9>50000thenprint"{down}line overlap with this prog!":goto50160
HM 50180 iffthen50290
AD 50190 forn=ba+atoba+9+a:a$(b)=mid$(str$(peek(n)),2)
MM 50200 iflen(a$(b))=1thena$(b)="  "+a$(b)
KN 50210 iflen(a$(b))=2thena$(b)=" "+a$(b)
FF 50220 ifn=eathenn=ba+9+a:next:f=1:goto50240
DI 50230 b=b+1:next:b=b-1
IA 50240 print"{clr}{white}"mid$(str$(l),2)" data";:forp=0tob-1:printa$(p)",";
KM 50250 next:printa$(p)
ID 50260 print"ba="ba":ea="ea":a="a":a=a+10:b=0:l="l":l=l+10:f="f":goto50180"
AJ 50270 print"{down}{down}{down}{blue}  line #"l"{white}"
FL 50280 poke631,19:poke632,13:poke633,13:poke634,31:poke198,4:end
HL 50290 print"{clr}";l;"forx="mid$(str$(ba),2)"to"mid$(str$(ea),2)":ready:";
FN 50300 print"pokex,y:next":print"sys53000"
KK 50310 print"{blue}ok: data entered...{down}{down}{down}{down}{down}{white}"
DD 50320 goto50280
```

Listing 2: *deleter.src*

```
10  ;block delete: lines >= 50000
20  ;line format:
30  ;   [link][line#][line data]0
40  ;   end of prog: 0 0 0 [end address]
50  ;        (link points to ^ )
60  ;
70  line       equ 50000    ;to delete
80  ;
90             org 53000    ;start address
100 ;note:  relocatable anywhere
110            ent          ;sys
120 ;
130 ;-- scan basic:
140 start      lda #01      ;link lb
150            sta $fb
160            lda #$08     ;link hb
170            sta $fc      ;ind.addr.
180 ;
190 ;scan link address:
200 scan       ldy #0       ;init.loop
210            lda ($fb),y  ;ld lnk lb
220            sta $fd      ;store it.
230            iny
240            lda ($fb),y  ;ld lnk hb
250            sta $fe      ;store it.
260 ;
270 ;scan line number:
280 line.no.   iny          ;next byte
290            lda ($fb),y  ;ld lne lb
300            cmp #<line   ;lb 50000?
310            bne link     ;n:nxt lnk
320            iny          ;y:nxt byt
330            lda ($fb),y
340            cmp #>line   ;hb 50000?
350            bne link     ;n:nxt lnk
360 ;
370 ;terminating zeros:
380 terminate dey           ;.y-2
390            dey
400            lda #0       ;store 0
410            sta ($fb),y  ; there.
420            dey          ; -1 byt
430            sta ($fb),y  ;again.
440 ;
450 ;move e.a. ptrs to byte after 000
460 move.ea    iny
470            iny          ;  +2 byts
480            tya
490            clc
500            adc $fb      ;lbfrm 251
510            sta 45       ;lb of ea
520            lda $fc      ;hbfrm 252
530            sta 46       ;hb of ea
540            rts
550 ;
560 link       ldx $fd      ;nxt lnklb
570            stx $fb      ;store it.
580            ldx $fe      ;      hb
590            stx $fc      ;ditto
600            lda #0
610            beq scan     ;=jmp scan
```

When Giants Walk...

Larry Rutledge, Sacramento, CA

Imagine how the Lilliputians felt. Let's face it, Gulliver was a *big* guy. Brobdingnagian, you might say.

No, we're not going to tell you. You'll have to type this in for yourself.

```
NP  0 rem screen display - larry rutledge
PI  1 rem transactor 9-5
JN  2 rem all rights reserved
MA  3 print chr$(147)
FN  4 print tab(11);chr$(154);chr$(17);
      chr$(17);"watch what happens"
ML  5 for i=1 to 1000:next
KH  6 for j=0 to 31:poke 53270,j:next
NG  7 get a$:if a$="" then 6
DK  8 poke 53270,200
```

The ML Column

Big numbers

by Todd Heimarck

This instalment of *The ML Column* started with one large idea that gradually developed into a series of smaller ideas. If a high-level language is like a pile of bricks from which you build a house, then machine language is like a pile of clay from which you make the bricks to build a house. It turned out that I needed some bricks.

The idea behind the original program, which remains in the planning stage, was to build an enormous look-up table within the 1750 RAM expansion unit. That's 512 kilobytes (or, in the program I had in mind, 4 megabits). With two bytes, you can count up to 65,535, which is not nearly high enough. The program needs several bytes to count up to four million.

Handling multi-byte numbers isn't so difficult, but there are two problems: input and output. The program needs a routine that accepts big numbers and turns them into binary values in memory. Plus, when the program is finished doing what it does, it needs a routine to convert the ones and zeros into printable ASCII numbers.

That topic is enough for a column. We'll have to discuss the RAM expander in some future column.

If you examine the beginning of Program 1, you'll see the essential structure of the program. It does five things:

1. Get a string from the user.
2. Convert it to a big (six-byte) binary value.
3. Do something with the number.
4. Convert it back to decimal.
5. Print the results.

At various spots along the way, the program prints appropriate prompts. It also checks for a zero value (the signal to exit the program) and for a number that's too big.

Tearing apart an ASCII number

I decided, for various reasons, to use a six-byte value and to store the low byte first. The binary number is stored in a section of memory I've named BIGSIX. It can hold numbers in the range 0-281,474,976,710,655 (hexadecimal $FFFFFFFFFFFF).

The user types on the keyboard, which means the incoming characters are ASCII values. If he or she types **910**, we'll receive 57, 49, and 48, because those are the ASCII codes for the characters '9', '1', and '0'.

The program accepts commas because it filters out any characters outside the range of ASCII numbers (48-57). If the user types **13turtle56**, the program stores the characters '1356' in memory. The memory buffer for the filtered characters is called MEMBUF.

We must deal with two special characters that might come along. An ASCII 13 is a carriage return character, which means the user pressed Return and is done. The loop ends when it sees a 13. In addition, a period can also mark the end of input. For example, if the user types **156.27**, we take that to mean 156 and not 15,267.

The first big brick, then, is the GSTRING subroutine. It repeatedly calls the Kernal routine CHRIN (which is preferable in this case to GETIN). It filters out all the ASCII numbers and stores them in MEMBUF.

Next, we call MAKEBIN, which converts the ASCII numbers into a six-byte integer. The process is relatively simple:

1. Start with a 0 in BIGSIX.
2. Multiply BIGSIX by 10 (because we're in base ten).
3. Get an ASCII number and subtract 48, to make the character '2' into the value $02 (or whatever).
4. Add that number to BIGSIX.
5. If there are more characters in MEMBUF, go back to step 2.

Say the user types **5993**. Start at the left. Zero times ten is 0. Add the first number (5). Five times ten is 50 and add 9, which is 59. Times 10 (590), plus 9 (599), times ten (5990), and add 3 (5993). It seems kind of silly to start with 5993 and end with it, but that's because the example math is in decimal. Inside the computer, it's all ones and zeros: 101 (5) becomes 110010 (50), 111011 (59), and so on.

A general routine for multiplying isn't necessary, because the only multiplication in the program involves the number ten. If you shift a binary number to the left, it's the same as multiplying by two. Shifting three times is equivalent to multiplying by eight. If x is the number, we want $10*x$. That's the same as $(2*x) + (8*x)$. To multiply by ten, shift left and temporarily save it (times two). Shift left twice more (times eight) and add the temporary value.

The BIGX10 routine multiplies BIGSIX by ten as part of the MAKEBIN routine. The ROTSIX subroutine rotates all six bytes one bit to the left, and the DUPSIX subroutine copies the BIGSIX number to BIG2. These routines also have to check for an overflow condition, which happens when the user types in a number bigger than 281 trillion.

Do something interesting

The first version of this program converted the ASCII numbers to binary and then back to ASCII, which is a rather pointless exercise. You type in a number and then it tells you the number you just typed. So what?

Since we've gone to all this trouble, we should do something. Program one takes the binary number and prints it as a series of ones and zeros. It happens in the PROCESS routine, which should be easy to follow.

A second version of PROCESS calculates the square root of the BIGSIX number. More about that in a moment.

Converting back to ASCII

After processing the binary value in PROCESS, we need the routine that converts back to printable base-ten numbers. Since we multiplied by ten in the other routine, it's a good bet that we need to divide by ten in the MAKEDEC routine. Since we started at the left before, we probably need to start at the right.

Take a short number, like decimal 57 (binary 111001). Let's see, 111001 divided by 1010 is about 101 with a remainder of 111. In decimal, that means 57 divided by 10 is 5 with a remainder of 7. The value 7 can be added to 48 to get 55, which is the ASCII character '7'.

The MAKEDEC routine builds up a decimal (ASCII) number by following these steps:

1. Divide by 10.
2. Add 48 to the remainder to get an ASCII number.
3. Repeat until the result is 0 (with a remainder of 0-9).

A previous column discussed binary division; I won't repeat myself here. See *The ML Column* in Volume 9, Issue 2 of *Transactor* if you're interested in the details of binary division.

An important thing to remember is that shifting is radix-dependent (if that's a word), whereas both division and multiplication are radix-independent. The decimal (base-ten) number 578 shifted to the left is 5780. The binary (base-two) number 101 shifted left is 1010. In base ten, shifting left is the same as multiplying by 10. In base two, shifting left is the same as multiplying by 2.

To find out that the rightmost digit of 914 is 4, we must actually divide by 10 (and get the remainder of 4). We cannot shift right, because the number is stored as a binary quantity. Shifting a binary right is equivalent to dividing by 2, not dividing by 10.

Calculating big square roots

Program 2 is a code fragment that contains a new PROCESS routine. Its lines replace the lines from Program 1; the first part is identical.

This new PROCESS routine finds the binary square root of a big six-byte number, rounded down to the nearest integer. It will say that the square root of 25 is 5, but it also says that the square root of 30 is 5.

Sounds complicated, doesn't it? It's not. If you're curious about how it works, read on.

Let's begin with a high-school math refresher and take a square root entirely in base ten. The square root of 65,536 is 256. The process is described below and corresponds to the representation included as Figure 1.

```
 2       5       6
)6   .  55   .  36
 4                      (0+2)*2
 2      55
 2      25              (40+5)*5
        30      36
        30      36      (500+6)*6
                 0
```

Figure 1: *Finding the square root of 65,536*

First, mark off every two digits starting at the decimal point: 6/55/36. Start with a partial answer of 0 because we haven't begun. Multiply by 20 to get 0 (call this the weird number). Look at the first number from the left. It's a 6 (the first clump from 6/55/36). Write down 2 as a partial answer, because (2 plus weird) times 2 is 4, which fits into 6. Subtract 4 from 6 (2) and append the next two numbers (55). Now we've got 255.

The new weird number is the partial answer (2) times 20, which is 40. We need a number x, where $(40+x)*x$ fits into 255. Five will do, because 45*5 is 225 and 46*6 is 276 (too big). The new partial answer is 25, the new weird number is 500, and the new target is 3036. The final digit in the answer is 6, because 506*6 is 3036, which exactly equals 3036.

We need to translate that to base two. As it turns out, the formula for figuring out the weird number is not "times twenty," it's "times two, then shift left". Remember the difference?

In base two, we need to multiply by 4 instead of 20. No problem. Just shift left twice.

What about guessing which number is next? Well, there's only two choices: zero or one. So let's say the weird number is 110100. If the answer is one, then we subtract 110101 (because of the rule to add the digit to the weird number).

It works out very nicely in binary. Shift the partial answer twice to the left and add one. If that fits into the number below, the next digit in the partial answer is 1. Otherwise, it's a zero.

That's the end of this column about big six-byte numbers, but it's a good start on another program that will do more with big numbers. We've got a few bricks now.

Listing 1: *BIG1.SRC*

```
DN 100 rem save"big1.src",8
PD 110 sys700
IL 120 *=49152
KO 130 .opt oo
DO 140 za      = $fb          ;  free zero-page location
NL 150 temp    = $fd
FH 160 temp2   = $fe
CG 170 counter = $ff
LI 180 errflag = $02
CK 190 chrin   = $ffcf
PO 200 chrout  = $ffd2
LC 210 ;  this is the main loop
GJ 220 main    lda #0
DO 230         sta errflag    ;  no errors (yet)
GL 240         jsr pmsg1      ;  print message 1
JB 250         jsr gstring    ;  get a string from user
BK 260         jsr makebin    ;  make it a binary number
FF 270         bcs error      ;  the number was too big
CO 280         jsr pmsg2      ;  print message 2
OA 290         jsr process    ;  do something to the number
MM 300         jsr makedec    ;  make the binary value into an ascii string
EA 310         jsr pmsg3      ;  print message 3
LD 320         jmp main       ;  loop forever
NG 330 error   bne realerr    ;  if not equal, a real error occurred
HG 340         rts            ;  else, it was a zero entry
JL 350 realerr jsr pemsg      ;  print error message "too big"
PK 360         jmp main       ;  and go back
OB 370 ;  generic loop for printing a message.
OP 380 msgout  = *
IA 390         sta za
PN 400         stx za+1       ;  store the address
MA 410         ldy #0         ;  begin at the beginning
KI 420 msglp   lda (za),y     ;  get the character
OO 430         beq msgex      ;  exit if zero
KG 440         jsr chrout     ;  else print it
CK 450         iny
CN 460         bne msglp      ;  continue loop
HE 470 msgex   rts            ;  end of msgout
CC 480 pmsg1   = *
HL 490         lda #<msg1
FB 500         ldx #>msg1     ;  the address
LI 510         jmp msgout     ;  implied rts at the end
DA 520 msg1    .asc "enter a number (commas ok).
FI 530         .byte 13,0     ;  end of pmsg1
AG 540 pmsg2   = *
FP 550         lda #<msg2
HF 560         ldx #>msg2
BP 570         jmp msgout
IO 580 msg2    .byte 13
FD 590         .asc "calculating...
OM 600         .byte 13,0      ;  end of pmsg2
IK 610 pmsg3   = *
ND 620         lda #<msg3
PJ 630         ldx #>msg3
GM 640         jsr msgout      ;  print, but then
OF 650         lda #<membuf
BO 660         ldx #>membuf    ;  print the buffer, too
PG 670         jsr msgout
OC 680         lda #13
JI 690         jsr chrout
DJ 700         jsr chrout
CL 710         rts
FH 720 msg3    .byte 13
BL 730         .asc "the answer is
PF 740         .byte 32,0      ;  end of pmsg3
KI 750 pemsg   lda #<errmsg
JF 760         ldx #>errmsg
JL 770         jmp msgout
PO 780 errmsg  .byte13
PF 790         .asc "** number is too big **
DO 800         .byte 13
KM 810         .asc "maximum is 281,474,976,710,655.
LN 820         .byte 13,0      ;  end of pemsg
BK 830 ;  the gstring routine gets a string and puts it in membuf
GD 840 ;  chrin sends a 13 to indicate the end, but we make it a zero
NB 850 gstring = *
HK 860         ldy #0
AD 870 gstlp   jsr chrin
PI 880         cmp #13         ;  if 13, we're done
GG 890         beq gstex
EH 900         cmp #46         ;  special case of period (53.15, for example)
KH 910         beq gstex
FH 920         jsr chknum      ;  better make sure it's a number
DB 930         bcs gstlp       ;  carry set means not-a-number
KE 940         sta membuf,y    ;  if it is a number, store it
GJ 950         iny
GN 960         bne gstlp       ;  branch always
BK 970 gstex   lda #0
HP 980         sta membuf,y    ;  store a zero
ON 990         rts             ;  store the length and exit
AN 1000 chknum cmp #58         ;  57 is ascii 9
OC 1010        bcs chkex       ;  if carry set, it's >"9", so exit w/carry set
GI 1020        cmp #48         ;  48 is ascii 0
KC 1030        bcc chkerr      ;  if carry clr, it's smaller than "0"
GI 1040        clc
PB 1050        rts             ;  clear carry = ok
BM 1060 chkerr sec             ;  set = nok
PH 1070 chkex  rts             ;  end of gstring
MJ 1080 ;  makebin makes the ascii numbers in membuf
MJ 1090 ;  into a 6-byte (48-bit) number by repeatedly multiplying by 10.
LL 1100 makebin = *
FA 1110         jsr clrbig      ;  clear out the big number
ME 1120         ldy #0          ;  start at the beginning
PN 1130 maklp   lda membuf,y    ;  get an ascii character  (48-57)
AI 1140         beq makend      ;  if zero, end of buffer
NC 1150         sta temp2       ;  else stash it
JG 1160         jsr bigx10      ;  and multiply bigsix by 10
EJ 1170         lda errflag
DK 1180         bne abort       ;  if error, then quit
BB 1190         lda temp2       ;  else get the digit back
BD 1200         sec
LH 1210         sbc #48         ;  make it 0-9
KD 1220         clc
GB 1230         adc bigsix
MG 1240         sta bigsix
NN 1250         bcc mak2        ;  skip ahead if no carry
EC 1260         inc bigsix+1
JI 1270         bne mak2        ;  else handle the higher bytes
MD 1280         inc bigsix+2
EA 1290         bne mak2
```

```
EF  1300          inc bigsix+3
IB  1310          bne mak2
MG  1320          inc bigsix+4
MC  1330          bne mak2
EI  1340          inc bigsix+5
AE  1350          bne mak2
MI  1360 abort    iny
LN  1370          sec
GC  1380          rts            ;  clear z flag and c flag to mark an error
JA  1390 mak2     iny
FF  1400          bne maklp      ;  branch always
JG  1410 makend   jsr bigor      ;  see if it's zeros
KC  1420          bne makex
HB  1430          sec
LO  1440          rts            ;  carry set = error/exit
EG  1450 makex    clc
KG  1460          rts            ;  clear carry means all is well
IH  1470 clrbig   ldx #5
DL  1480          lda #0
DD  1490 clrlp    sta bigsix,x   ;  clear all six bytes
OC  1500          dex            ;  count back
OK  1510          bpl clrlp
MN  1520          rts
KB  1530 bigx10   jsr rotsix     ;  rotate bigsix to the left (times 2)
IA  1540          jsr dupsix     ;  copy it to big2
EB  1550          jsr rotsix
OB  1560          jsr rotsix
JB  1570          lda #6
OK  1580          sta temp       ;  do all six bytes
NA  1590          ldx #0         ;  starting with byte 0
GL  1600          clc
PI  1610 bigxlp   lda big2,x     ;  add 2x
LA  1620          adc bigsix,x   ;  to 8x
BE  1630          sta bigsix,x   ;  which equals 10x
PH  1640          inx            ;
KN  1650          dec temp
LO  1660          bne bigxlp     ;  add all six bytes
EI  1670          bcc noflow     ;  if cc, no overflow
GD  1680          inc errflag    ;  else set error flag
KP  1690 noflow   rts            ;  return from bigx10
DH  1700 rotsix   asl bigsix
JK  1710 rot2six  rol bigsix+1   ;  rotate bigsix one bit left
NC  1720          rol bigsix+2
LD  1730          rol bigsix+3
JE  1740          rol bigsix+4
HF  1750          rol bigsix+5
IN  1760          bcc ok         ;  if cc, no overflow
GK  1770          inc errflag    ;  set error flag
FL  1780 ok       rts
DM  1790 dupsix   ldx #5         ;  copy bigsix to big2
AG  1800 duplp    lda bigsix,x
FA  1810          sta big2,x
DN  1820          dex
IC  1830          bpl duplp
MB  1840          rts
LC  1850 bigor    lda bigsix+0
CK  1860          ora bigsix+1
PK  1870          ora bigsix+2
ML  1880          ora bigsix+3
JM  1890          ora bigsix+4
GN  1900          ora bigsix+5
EC  1910          rts            ;  end of makebin
CP  1920 ;  makedec turns the binary value into
FA  1930 ;  a series of ascii numbers in membuf
PM  1940 makedec = *
JI  1950          lda #0
JH  1960          pha            ;  start with 0, which means "end"
IA  1970 mdlp1    ldx #48
OB  1980          stx counter    ;  48 bits
BL  1990          lda #0
LH  2000          sta temp
LL  2010 mdlp2    jsr rotsix     ;  get a bit
BM  2020          rol temp       ;  and move it into temp
LF  2030          lda temp
FO  2040          cmp #10        ;  if temp < 10
FF  2050          bcc mdcool     ;  it's cool, don't worry
BJ  2060          sbc #10        ;  else subtract 10 (carry is already set)
DP  2070          sta temp       ;  store it (carry is still set)
LI  2080 mdcool   php
BP  2090          lsr bigsix
IP  2100          plp
JA  2110          rol bigsix     ;  put the bit back into bigsix
HJ  2120          dec counter
PP  2130          bne mdlp2
JM  2140          lda temp
HE  2150          ora #48        ;  make it ascii
HN  2160          pha            ;  and push it
FP  2170          jsr bigor      ;  is it zero yet
PE  2180          bne mdlp1      ;  no, it isn't.  go back.
JN  2190          ldy #0
JK  2200 mdlp3    pla            ;  get a character
HN  2210          sta membuf,y
KA  2220          beq mdex       ;  if 0, we're done
EA  2230          iny            ;  else increment y
JL  2240          bne mdlp3      ;  and branch always
DC  2250 mdex     rts            ;  end of makedec
LP  2260 ;  process prints the 1s and 0s of the binary number
HP  2270 process lda #<binmsg
IB  2280          ldx #>binmsg
DM  2290          jsr msgout
IE  2300          ldy #5         ;  six bytes
FD  2310 proclp1 ldx #8          ;  eight bits
OP  2320          stx counter
IF  2330          lda bigsix,y
PM  2340          sta temp
AB  2350 proclp2 lda #48         ;  ascii zero
OP  2360          rol temp
ME  2370          bcc sendit     ;  if zero, print "0"
PC  2380          adc #0         ;  else add 0 (plus set carry)
LM  2390 sendit   jsr chrout
PK  2400          dec counter
LF  2410          bne proclp2
KP  2420          lda #13
KO  2430          jsr chrout     ;  next line
DE  2440          dey
CC  2450          bpl proclp1    ;  continue outer loop
II  2460          rts
LB  2470 binmsg   .asc "binary (high byte to low):
GA  2480          .byte 13,0     ;  end of process
OA  2490 bigsix   = $c400        ;  th* big 48-bit (6-byte) number
JI  2500 big2     = $c406        ;  another big number
EI  2510 membuf   = $c40c        ;  memory buffer
```

Listing 2: *BIG2.SRC - To create this program, use lines 100-2250 of BIG1.SRC and add the following lines for the new PROCESS routine.*

```
JD  2260 ; process finds the square root.
BP  2270 big4     .byte 0,0,0,0,0,0
KP  2280 big3     .byte 0,0,0,0,0,0
AO  2290 process = *
AE  2300          ldx #5         ;  first copy bigsix to big4 and clear big3
CL  2310 proclp1 lda bigsix,x
LA  2320          sta big4,x
FA  2330          lda #0
LB  2340          sta big3,x
FO  2350          dex
IE  2360          bpl proclp1
DO  2370          jsr clrbig     ;  clear out bigsix for the answer
HN  2380          lda #24
IE  2390          sta counter    ;  repeat the loop 24 times (for 48 bits)
OC  2400 pmain    = *            ;  main loop for process
KE  2410          jsr dupsix     ;  copy bigsix to big2
KO  2420          clc
AF  2430          jsr rot2       ;  rotate big2
JA  2440          sec
DD  2450          jsr rot2       ;  rotate again plus 1
HE  2460          jsr rot43      ;  rotate big4 into big3 twice
II  2470          jsr comp32     ;  compare big3 to big2
NF  2480          php            ;  save the processor status
JG  2490          rol bigsix
HC  2500          jsr rot2six    ;  put the bit into bigsix
CJ  2510          plp
```

```
CB  2520          bcc nosub     ;  if carry is clear, continue
JC  2530          ldx #0
AN  2540          ldy #6        ;  else subtract 3-2
GL  2550 pmlp     lda big3,x
FM  2560          sbc big2,x
BA  2570          sta big3,x
AP  2580          inx
JN  2590          dey
IL  2600          bne pmlp
HN  2610 nosub    dec counter
NN  2620          bne pmain
CD  2630          rts
EE  2640 ;  rot2 rotates big2 to the left
EL  2650 rot2     ldx #0
LL  2660          ldy #6
GD  2670 r2lp     rol big2,x
EF  2680          inx
ND  2690          dey
IO  2700          bne r2lp
CI  2710          rts
MA  2720 ;  rot43 rotates two bits from big4 into big3
DP  2730 rot43    jsr twice     ;  do it twice
EH  2740 twice    ldx #0
BK  2750          ldy #12
NE  2760 r43lp    rol big4,x
OK  2770          inx
HJ  2780          dey
LD  2790          bne r43lp
MN  2800          rts
BJ  2810 ;  comp32 compares big3 to big2
IG  2820 comp32   ldx #5        ;  compare msb's first
EC  2830 c32lp    lda big3,x
CB  2840          cmp big2,x
EA  2850          bcc c32ex     ;  if cc, big2>big3
NL  2860          bne c32ex     ;  if not equal, big3>big2
NO  2870          dex
MA  2880          bpl c32lp     ;  else they're equal, so go back
PL  2890 c32ex    rts
IK  2900 bigsix   = $c400       ;  th* big 48-bit (6-byte) number
DC  2910 big2     = $c406       ;  another big number
OB  2920 membuf   = $c40c       ;  memory buffer
```

Listing 3: *BIG1.GEN - BASIC generator for "big1.obj"*

```
JP  100 rem  generator for "big1.obj"
HN  110 n$="big1.obj": rem  name of program
BC  120 nd=568: sa=49152: ch=60233
```

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

```
NF  1000 data 169,   0, 133,   2,  32,  56, 192,  32
AO  1010 data 220, 192,  32,   6, 193, 176,  15,  32
MH  1020 data  92, 192,  32, 242, 193,  32, 182, 193
FF  1030 data  32, 116, 192,  76,   0, 192, 208,   1
PB  1040 data  96,  32, 155, 192,  76,   0, 192, 133
CC  1050 data 251, 134, 252, 160,   0, 177, 251, 240
NC  1060 data   6,  32, 210, 255, 200, 208, 246,  96
CK  1070 data 169,  63, 162, 192,  76,  39, 192,  69
IL  1080 data  78,  84,  69,  82,  32,  65,  32,  78
BL  1090 data  85,  77,  66,  69,  82,  32,  40,  67
CO  1100 data  79,  77,  77,  65,  83,  32,  79,  75
GK  1110 data  41,  46,  13,   0, 169,  99, 162, 192
BE  1120 data  76,  39, 192,  13,  67,  65,  76,  67
OO  1130 data  85,  76,  65,  84,  73,  78,  71,  46
HM  1140 data  46,  46,  13,   0, 169, 139, 162, 192
JJ  1150 data  32,  39, 192, 169,  12, 162, 196,  32
IN  1160 data  39, 192, 169,  13,  32, 210, 255,  32
DA  1170 data 210, 255,  96,  13,  84,  72,  69,  32
NC  1180 data  65,  78,  83,  87,  69,  82,  32,  73
EO  1190 data  83,  32,   0, 169, 162, 162, 192,  76
GH  1200 data  39, 192,  13,  42,  42,  32,  78,  85
FC  1210 data  77,  66,  69,  82,  32,  73,  83,  32
CD  1220 data  84,  79,  79,  32,  66,  73,  71,  32
GD  1230 data  42,  42,  13,  77,  65,  88,  73,  77
KD  1240 data  85,  77,  32,  73,  83,  32,  50,  56
ND  1250 data  49,  44,  52,  55,  52,  44,  57,  55
JE  1260 data  54,  44,  55,  49,  48,  44,  54,  53
LA  1270 data  53,  46,  13,   0, 160,   0,  32, 207
JF  1280 data 255, 201,  13, 240,  15, 201,  46, 240
IE  1290 data  11,  32, 250, 192, 176, 240, 153,  12
GI  1300 data 196, 200, 208, 234, 169,   0, 153,  12
JF  1310 data 196,  96, 201,  58, 176,   7, 201,  48
KI  1320 data 144,   2,  24,  96,  56,  96,  32,  79
LE  1330 data 193, 160,   0, 185,  12, 196, 240,  54
BF  1340 data 133, 254,  32,  90, 193, 165,   2, 208
NJ  1350 data  39, 165, 254,  56, 233,  48,  24, 109
CG  1360 data   0, 196, 141,   0, 196, 144,  28, 238
GG  1370 data   1, 196, 208,  23, 238,   2, 196, 208
CL  1380 data  18, 238,   3, 196, 208,  13, 238,   4
IL  1390 data 196, 208,   8, 238,   5, 196, 208,   3
CP  1400 data 200,  56,  96, 200, 208, 197,  32, 163
GN  1410 data 193, 208,   2,  56,  96,  24,  96, 162
BA  1420 data   5, 169,   0, 157,   0, 196, 202,  16
PB  1430 data 250,  96,  32, 128, 193,  32, 151, 193
PO  1440 data  32, 128, 193,  32, 128, 193, 169,   6
NL  1450 data 133, 253, 162,   0,  24, 189,   6, 196
PO  1460 data 125,   0, 196, 157,   0, 196, 232, 198
JK  1470 data 253, 208, 242, 144,   2, 230,   2,  96
AL  1480 data  14,   0, 196,  46,   1, 196,  46,   2
DJ  1490 data 196,  46,   3, 196,  46,   4, 196,  46
ME  1500 data   5, 196, 144,   2, 230,   2,  96, 162
JC  1510 data   5, 189,   0, 196, 157,   6, 196, 202
LL  1520 data  16, 247,  96, 173,   0, 196,  13,   1
JI  1530 data 196,  13,   2, 196,  13,   3, 196,  13
DG  1540 data   4, 196,  13,   5, 196,  96, 169,   0
FE  1550 data  72, 162,  48, 134, 255, 169,   0, 133
NG  1560 data 253,  32, 128, 193,  38, 253, 165, 253
EC  1570 data 201,  10, 144,   4, 233,  10, 133, 253
PE  1580 data   8,  78,   0, 196,  40,  46,   0, 196
LP  1590 data 198, 255, 208, 229, 165, 253,   9,  48
LC  1600 data  72,  32, 163, 193, 208, 211, 160,   0
DN  1610 data 104, 153,  12, 196, 240,   3, 200, 208
CN  1620 data 247,  96
AJ  1630 rem -- following data is for process1 --
MM  1640 rem -- prints input number in binary  --
IG  1650 data 169,  28, 162, 194,  32,  39
CM  1660 data 192, 160,   5, 162,   8, 134, 255, 185
PJ  1670 data   0, 196, 133, 253, 169,  48,  38, 253
DJ  1680 data 144,   2, 105,   0,  32, 210, 255, 198
OK  1690 data 255, 208, 241, 169,  13,  32, 210, 255
BD  1700 data 136,  16, 224,  96,  66,  73,  78,  65
JA  1710 data  82,  89,  32,  40,  72,  73,  71,  72
KD  1720 data  32,  66,  89,  84,  69,  32,  84,  79
KN  1730 data  32,  76,  79,  87,  41,  58,  13,   0
```

Listing 4: *BIG2.GEN - BASIC generator for "big2.obj"*

```
MP  100 rem  generator for "big2.obj"
IN  110 n$="big2.obj": rem  name of program
EC  120 nd=625: sa=49152: ch=68145
```

(for lines 130-260, see the standard generator on page 5)
(use lines 1000-1620 of "big1.gen", change the 242 in
line 1020 to 254 and add the following lines)

```
DM  1630 rem -- following data for process2 --
MK  1640 rem -- finds square root of number --
IH  1650 data   0,   0,   0,   0,   0,   0
OG  1660 data   0,   0,   0,   0,   0,   0, 162,   5
JK  1670 data 189,   0, 196, 157, 242, 193, 169,   0
EE  1680 data 157, 248, 193, 202,  16, 242,  32,  79
MO  1690 data 193, 169,  24, 133, 255,  32, 151, 193
IM  1700 data  24,  32,  70, 194,  56,  32,  70, 194
PG  1710 data  32,  82, 194,  32,  97, 194,   8,  46
II  1720 data   0, 196,  32, 131, 193,  40, 144,  17
BA  1730 data 162,   0, 160,   6, 189, 248, 193, 253
IO  1740 data   6, 196, 157, 248, 193, 232, 136, 208
OJ  1750 data 243, 198, 255, 208, 208,  96, 162,   0
CC  1760 data 160,   6,  62,   6, 196, 232, 136, 208
DO  1770 data 249,  96,  32,  85, 194, 162,   0, 160
FP  1780 data  12,  62, 242, 193, 232, 136, 208, 249
KB  1790 data  96, 162,   5, 189, 248, 193, 221,   6
JA  1800 data 196, 144,   5, 208,   3, 202,  16, 243
LJ  1810 data  96
```

Transactor Reader Survey

Here it is! Transactor's first reader survey. We want to know what you like and don't like about Transactor. We want to know what software and hardware you're using. And we'd like to know what you want to see in the magazine in the future. Nosey, aren't we?

Please take the time to do the survey and send us the results. Make a photocopy. (We wouldn't want you to tear up the magazine). Please send your completed survey page to: **Reader Survey, *Transactor*, 85 West Wilmot St., Unit 10, Richmond Hill, Ontario, Canada, L4B 1K7**.

System configuration and software used: ____________________

____

____

____

		Love it!	Like it.	Don't like.	Hate it!
Columns	The ML Column	❑	❑	❑	❑
	The Edge Connection	❑	❑	❑	❑
Features	Bits	❑	❑	❑	❑
	Letters	❑	❑	❑	❑
	News	❑	❑	❑	❑
	Reviews	❑	❑	❑	❑

		Love it!	Like it.	Don't like.	Hate it!
Article Topics					
	C64 Native Mode	❑	❑	❑	❑
	C128 Native Mode	❑	❑	❑	❑
	Power C	❑	❑	❑	❑
	CP/M	❑	❑	❑	❑
	GEOS	❑	❑	❑	❑
	Reference Material	❑	❑	❑	❑
	Hardware Projects	❑	❑	❑	❑
	Software Reviews	❑	❑	❑	❑
	Hardware Reviews	❑	❑	❑	❑
	Theoretical Material	❑	❑	❑	❑

Comments or suggestions: ____________________

____

____

____

The Edge Connection

GEOS 128 2.0, ZOOM, macros, radio, etc.

by Joel Rubin

The most important recent 8-bit Commodore software news is the release of GEOS **2.0** for the C128. Like GEOS **2.0** for the C64, it includes a new version of **geoWrite Workshop** (including *Text Grabber*, *geoMerge*, and *geoLaser*), *geoSpell* (but not the font editor which came with the separately packaged *geoSpell* and is now part of **Fontpack Plus**), a new deskTop (with a built-in clock for midnight hackers, and, more important, the ability to select more than one file for copying or erasing), et alia. The new versions of *geoWrite* and *geoSpell* only work in 80-column mode.

In a WIMP (Windows Icons Mouse Pointer) system such as GEOS, as you become an expert, you may start wishing for cryptic control codes or a command line interface. These may be 'user unfriendly' but at least they get the job done without going through menu after menu. Both the GEOS deskTop and GEOS applications are gradually moving towards control code alternatives, with each version of GEOS having more and more of them.

I still think I would generally prefer to use *Paperclip* or a similar character-based post-formatted word processor to enter text and then use *Text Grabber* to convert to *geoWrite*, if I'm not satisfied with an ASCII/control-code printout. But (especially with the 2 MHz clock) *geoWrite* is becoming less of a pain in the neck as a text editor. Still, there are times when I want to move a margin and get a tab marker instead, or when I want to enter text and accidentally move the mouse and find things scrolling ever so slowly the other way. At those times, I think that the best game to play with a mouse is that canine favourite, "Shake It To Death".

There are quite a few new printer drivers, including some that work with a user port to Centronics cable (such as the one described in *Transactor* Volume 9, Issue 3), some double and quadruple strike drivers, a few drivers that reduce the size of the page and offer greater dot density, and even a very few drivers that will work with RS-232 interfaces and a PAL clock speed. (Question: How do the Commodore computers work in France, where they have neither the NTSC nor PAL colour standards but SECAM?)

Unfortunately, this new version of GEOS does not take advantage of the 128's colour capabilities in 80-column mode, so all 80-column work is going to be in mono. Also, as usual, *geoWrite* only supports an 8" (20 cm) wide page, so if you want to use something wider, you are out of luck. The other immediately obvious deficiency is the manual (which doesn't really exist - you get a copy of the 64 version manual and a booklet that goes through the 128 differences, chapter by chapter). I always thought that one of the advantages of using a computer for word processing was that it would be easier to edit one's document and come out with a new edition.

By the way, although **Geoprogrammer 2.0** is listed as an application one might own, I was told by customer service that this product has been cancelled. Since the current version of **Geoprogrammer** and, in particular, *GeoDebugger* only work under C64 GEOS, this leaves quite a gap. Readers might want to express their opinions. Leave them on Q-Link, or write them to Berkeley Softworks, 2150 Shattuck Avenue, Berkeley, CA 94704.

German viruses invade Michigan!

Now that we've gotten through with the supermarket tabloid-type headline, I can tell you that this is about a Data Becker book newly translated into English and released by Abacus - *Computer Viruses, A High Tech Disease*. On a scale of 1 to 10, I would give this book an 8 or 9. My major complaint would be that it could use a good bibliography. It deals calmly and rationally with the subject, unlike so much of the popular press. Technical points are illustrated with programs in various languages, including GW-BASIC, Turbo Pascal, IBM VM/CMS command language (the infamous "Christmas virus"), and MS-DOS and IBM mainframe assembly language. The Arpanet virus happened too late to be covered.

(Just in case you think that Commodore 8-bit machines are immune from viruses, there is a short listing of CP/M+ BDOS calls; although GEOS, which creates swap files, is far more vulnerable. Always keep a write-protect on original GEOS and GEOS application disks, unless you need to write on them - for example, to install the default printer driver. Some versions of GEOS will, in an anti-piracy trap, erase boot files if they 'think' you are using a copy, and the test might be sensitive to drive alignment.)

This is not just a technical hacking book. There are interviews with security professionals, such as a Bavarian police detective and the head of an insurance company. (By the way, those who are fighting for the good name of "hacker" will be glad to note that the word is used in this book as a synonym for "technically-oriented computer user, whether law-abiding or not".)

Of course, the legal references in this book are largely irrelevant to most readers of this magazine - not only are the specific laws different, but German law derives largely from Roman law and the *Code Napoléon*, whereas most of you live under some version of English Common Law. (Frankly, I think that it is a mistake to make too many specific laws when the old laws, such as "Malicious Mischief", still have teeth. When the legislators pass a lot of situation-specific laws, the codes get cluttered with laws, many of them technologically obsolete, and it becomes difficult for the average citizen to determine his or her rights and responsibilities.)

A commercial disk-loaded monitor for the C64 and C128

ZOOM is a machine language monitor for the C64, C128, and the BBC computer. (Has anyone outside Britain ever seen a BBC computer? I've seen books about programming them but that's it. It was a 6502-based computer put out by Acorn. *[Acorn computers have been available here in Toronto. - Ed.]* It had a very nice structured BASIC that included a built-in assembler, but that's about all I know about it.) Like most monitors available for Commodore computers, it is based on the monitor built into all but the earliest PET/CBM computers and the extensions written for it, such as *Supermon*, *Extramon* and *Micromon*.

The most important feature which this monitor has which is not in the built-in C128 monitor is the ability to walk (single step) code at three different speeds and to quick trace code (run code until you've passed a certain point **n** times and after that walk the code). One can also set up non-standard banks, such as $3E (BANK 0 with I/O), check memory for illegal opcodes, and enter values into memory as PETSCII values instead of hex. There is a built-in hex calculator and a wedge; I can't figure out how to make *ZOOM*'s wedge work for device 9.

Let's say you are looking through memory for text, but you are not certain whether it's PETSCII, ASCII, or screen codes. *ZOOM* allows you define a series of mask bytes so that you are looking for an area of memory where byte-1 and mask-1 = hunt-pattern-1, byte-2 and mask-2 = hunt-pattern-2, et alia. *ZOOM*, like *Extramon*, has a relocation facility so that you can change absolute addresses and/or tables of words to a different location. Of course, this doesn't always work. Relocation routines generally will not work if you have the following sort of code:

```
lda #<$8000
sta $61
lda #>$8000
sta $61+1
ldy #0
lda ($61),y
```

since the relocation routine will not find the actual absolute address $8000 anywhere. However, *ZOOM*'s relocation routines do work on *ZOOM* itself. The C64 version loads at $C0000, but you should be able to set it up to work at the top or bottom of BASIC. (I didn't get the C64 version. C64 users should note that there are several cheap or free alternatives, including *Micromon*, from public domain sources, and *HESMON*, available on cartridge from software liquidators.) The C128 version loads at $8000 and takes 6K of memory. It can be relocated to anywhere up to or below $A800-$BFFF. (It works either in memory configuration $0E or $4E - BANK 0 or BANK 1 RAM, BASIC switched out, KERNAL ROMs switched in, and I/O switched in. The very thin (16 pages for all three versions!) manual states that *ZOOM* can be relocated but it does not state where; or that it will work in either RAM bank. It uses a bit of interface code in common RAM around $0290.)

So far, ZOOM is the only single-step and/or quick trace routine I have been able to find for C128 mode. Abacus' *BASIC 7.0 Internals* book mentions that a certain author has written another one, and I have sent him two SASE's but have received no response in at least a year.

One other advantage that *ZOOM* has over the built-in C128 monitor is that it uses four hex digit addresses and displays the configuration register rather than five hex digit addresses. If you hit a BRK in a machine language program and get into the standard C128 monitor, the bank given as part of the register display may or may not have anything to do with reality - especially if you are playing with BASIC ROM routines and get dumped into one of those non-standard banks with which BASIC 7.0 is so enamoured.

ZOOM is marketed by Supersoft in the U.K. and the Commodore versions are imported by Skyles Electric Works (231-E S. Whisman Rd., Mountain View, CA, 94041, telephone (800) 227-9998/(415) 965-1735) in the U.S.

Assembly language macros

Xytec's *Macro Set 1* (1924 Divisidero St., San Francisco, CA, 94115, (415) 563-0660) is a disk of assembly language macros for use with Commodore's C64 Assembler (available cheap from software liquidators) or *Merlin*. While the *Merlin* macros, at least, work with the C128 version of *Merlin*, the object code generated may not work in C128 mode. For example, in C128 mode, before you open a file, you have to call SETBNK to tell the Kernal whether the name of the file is in BANK 0 or BANK 1. Generally, BASIC uses BANK 1, but most machine language programs use BANK 0, so you can't just assume the default is correct.

The macros are generally oriented to business, rather than graphics or games software. Among other things, there is a code for a sort of 16-bit virtual machine, somewhat similar to the machine defined by Steve Wozniak's *Sweet-16*, although *Sweet-16* is an interpreted language. There is BCD multibyte arithmetic, and output numeric formatting, professional-looking keyboard input and Kernal calls. There is some useful stuff in here if you have no trouble using disk files from BASIC but have trouble with the machine language calls for opening the files and checking the error channel, or if you want to read delimited variable-length records, similar to INPUT# in BASIC. One other useful bit of code allows you to create a self-debugging version of your program that will, at specified points, dump the registers or an area of memory, await a key press, and then restore the text screen and registers and continue. This disk costs $29.95 (U.S.).

Xytec, very much a 'garage-type' operation, also sells a few other programs, including a tree-oriented data base (intended, in part, for generating a disk-based user manual), a roommate accounting program, and a lotto number choosing program, which is, essentially, just a somewhat entertaining random number generator. (Xytec's lotto program does not "help you win the lotto by analyzing previous lotto drawings", but then, no statistician believes that this is mathematically possible. The so-called "law of averages" does not apply to independent events, such as the tossing of a coin or the drawing of lotto numbers. It does apply to Blackjack, since if the drawn card is not put back in the deck, the odds of getting cards with the same value is decreased.) Xytec's catalog is free; their demo disk is $2, applicable to an order. Currently, Xytec is willing to send you a disk now and ask you for the money, or the returned disk, later. The disks are not copy-protected. Is this trust justified? We shall see.

Mail order legal problems in the U.S.

The mail order industry in the U.S. is screaming about a bill in Congress. Some states have "use tax" laws that require you to pay sales tax when making purchases in another state to use in the taxing state. If you buy a car in another state, your state's vehicle licensing authority may have no trouble collecting the difference, if any, between the tax you paid and the tax in your home district when you go to register the car. In most cases, mail order firms do not collect taxes on out-of-state purchases, leaving the customer, who may not even know of the requirement, to obtain an obscure form and pay the tax.

The state tax collectors have asked Congress to force mail order firms to collect all customers' sales tax. This will raise mail order firm's expenses. What the mail order firms can't say is that it will also discourage interstate customers who have been trying to evade their tax. If a one-person garage operation has to keep track of all the state taxes - which frequently differ in different localities in the same state (I pay a 1/2 per-cent transit district surcharge on my state sales tax) - it may just drive that operation out of business. If you live in the U.S. and buy or sell mail order and have any opinion, pro or con, you might wish to contact your representative and senators.

Computers and radio

If you're interested in using a computer for shortwave listening or amateur radio, you might be interested in the free pamphlet, *INFODUTCH*, available from Radio Netherland's hobbyist show, *Media Network*. The address is Postbus 222, 1200 JG Hilversum, The Netherlands. Among the references given in the guide is one for the Commodore Radio Users Group, 22 Whiteford Avenue, Bellsmyre, Dumbarton G82 2JT, U.K., telephone 44 389 61250. In the U.K, their magazine costs £8 per year. If you write for information from overseas, send them return postage - say two International Reply Coupons, or British mint stamps.

Finally, here are a couple of quick tips. Let's say you want to poke to the C64 text screen in interpreted BASIC. You can avoid the extra step of poking to colour memory if you clear the screen using: **poke 53281,1:print chr$(5)chr$(147);:poke 53281,x** where **x** is either 0 or 2-15. Text will be in white. Exactly how this works depends on which model C64 you have.

On the oldest (Kernal 1) machines, printing chr$(147) ("clr") always clears colour memory to white. On middle-aged (Kernal 2) machines, chr$(147) clears colour memory to the background colour (peek(53281)). Thus, by poking 1 into the background before we clear the screen, we will get white colour memory.

Many old public domain programs and even a few commercial programs cause invisible (background-coloured) pokes on Kernal 2 machines. Kernal 3 machines, which includes newer 64s, all 64Cs, and the 64 mode of 128s, clear colour memory to the current text colour, which is set to white by chr$(5). The SX-64 uses a version of the Kernal 3 ROM, although it has certain differences - for example, the default colours are blue on white instead of light blue on blue, and trying to access the cassette unit gives an illegal device error.

There is one more official North American Kernal - the Educator model. This computer looks like a PET, with a metal case and built-in green-screen monitor, and it was sold by some liquidators. I haven't the foggiest idea what differences it has from other models except that Commodore/Terrapin LOGO looks for an identifier byte and gives you a special sign-on message if it is found. I don't have access to the European models but, since the North American Kernals contain PAL RS-232 code, and the European disassembly books seem to have the same code, I assume that this will work on overseas C64s also.

The second hint comes from Joe Dawson, President of Xytec Software. One easy method of setting up a text screen which appears to have different background colours in different areas is simply to use reverse characters throughout. For example, you could have a black background, a blue border, and two boxes filled with text. Everywhere outside the boxes is filled with reverse blue spaces, which appears to make the border extend around the boxes. The first box is filled with white reversed spaces and text which appears to be black text on a white background. The other box is filled with red reversed spaces and text which appears to be black text on a red background. No raster interrupts or multi-colour text or high resolution screens are required. Moreover, this method will work just as easily on the 80-column screen of a C128 without *BASIC 8* or 64K of video RAM.

Inner GEOS

A look at how GEOS operates

by William Coleman

Although GEOS allows you to quickly write programs that would takes weeks or months to write on a normal C64/128, few people actually program with it. Part of the problem is that most people don't understand how GEOS works. While the inner workings of GEOS are not hard to understand, they do function quite differently from what most 64/128 programmers are used to.

In this article, you will learn how GEOS functions and how its various levels interact with an application. Once you understand this interaction creating and debugging GEOS applications becomes much, much easier.

A 'normal' 64 or 128 program does most of the work itself and uses Kernal subroutines for things like disk I/O, printing characters, etc. GEOS is very different. Basically, it is composed of three levels:

Interrupt Level
The Main Loop
Application Routines

The first two levels are the GEOS Kernal itself and they must be allowed to execute periodically or GEOS will seem to freeze. In fact, if you are testing an application and everything seems to stop (mouse won't move, keyboard won't respond, etc.) you can be 90 per cent certain that your application went into an endless loop and is not letting GEOS do it's job.

Interrupt Level reads the keyboard and checks for numerous other conditions that require frequent, periodic inspection. It won't act on what it finds; there isn't enough time. So flags are set to indicate when the various conditions are met. The Main Loop then checks these flags and acts on them accordingly.

GEOS is an example of Event-Driven code. The Main Loop is simply a series of subroutine calls that operate in an endless loop. These subroutines check if certain events have occurred and if so execute Service Routines that the application has set up to handle those situations.

Service routines

There are basically three types of service routines. First, there are the routines that the application attaches to menus and icons. For example: when the user presses the mouse button, GEOS will check to see if the mouse is over a menu option or an icon. If one of these conditions ('events') is met, the Main Loop will call the service routine associated with that option/icon (defined in a menu or icon table).

The next type of service routine is called through the System Vectors. These are word length memory locations that contain the address of a service routine to execute when the condition associated with that vector occurs. If a vector contains $0000 then it will be ignored completely. In the above example, if the mouse is not over a menu or an icon, the Main Loop will check *otherPressVector*. If it is non-zero, the routine whose address is in the vector will be executed.

The third way to act on an event is through timers. These come in two flavors: Sleep and Processes.

Sleep

Sleep is a method of stopping the execution of a subroutine for a predetermined amount of time while continuing to execute the rest of the application. Remember, the Main Loop must have control periodically or GEOS will stop. Sleep provides a way to pause while still performing routine tasks.

The most noticable example is one that you see every time you use GEOS: icon and menu flashes. When a user clicks on a menu option, the Menu Handler will invert the option, put itself to sleep, and when it awakens, simply invert the option a second time.

Sleep works through a table of timers, one for each sleeping subroutine. Interrupt Level will decrement each timer every interrupt. The Main Loop checks these timers; and, if any have reached zero, the associated routine will be executed. When Sleep is called, the return address is pulled off the stack. This address is where execution will continue from.

Processes

GEOS has a form of multitasking called processes. A process is a subroutine that an application can set up that will be executed at regular intervals. The application passes the amount of time that should elapse between calls to the process. GEOS maintains timers for each process, just as it does for Sleep.

One point you must keep in mind: while the timers are updated during Interrupt Level, the execution of a process is done during Main Loop. This means that the time the Main Loop takes to go through one complete iteration can vary significantly depending on how many sleep and process routines need to execute and how long each one takes. Timer values less then about ten (1/6 of a second) will not be very accurate.

GEOS has no way to stack a process. If a process comes due before the previous call is done (i.e. a process times out twice before being executed), it will only be executed a single time.

OK, now that we know what events are and how they are used, let's take a look at the processing levels themselves.

Interrupt Level

Normally, a timer in one of the CIA chips generates interrupts that are used to read the keyboard, etc. GEOS is a bit different: sixty times per second a raster IRQ is generated by the VIC chip. This IRQ is set up to hit during vertical blanking so drawing to the screen will not produce flicker.

GEOS does considerably more during an interrupt then a regular 64 or 128; therefore, it doesn't have time to act on things like mouse presses and processes. That is the job of the Main Loop. The advantage of using interrupts to set flags and manage the timers is that the Main Loop is not constant. Interrupts provide a means to constantly monitor the system regardless of what else is going on.

Normally an application will not need to modify the interrupts but if the need arises there are two vectors provided for adding your own interrupt code. One, *intTopVector*, occurs before the bulk of the interrupt has occurred. The other, *intBotVector*, occurs at the end of the cycle.

Several things should be kept in mind when writing interrupt code:

- Don't try to do too much during interrupts. The GEOS interrupt is already quite long and you will bog down the application.
- Don't clear interrupts, i.e. CLI.
- Don't use GEOS routines if you can help it. Some will work during interrupts and some won't. *CallRoutine* and *DoInlineReturn* will work OK.

The Main Loop

This is where most of the nitty gritty parts of GEOS occur. Menus, processes, and the mouse are all managed from here (remember: interrupts read the status, the Main Loop acts on it). One of the biggest mistakes beginning GEOS programmers make is that they don't let the Main Loop have control.

The Figures that are included with this article are not byte by byte disassemblies (which change with time). They are presented only to give you an idea of what's going on...

An application is really little more then a group of service routines. They are called by the Main Loop and when they are finished they must RTS back to it. Since the Main Loop controls the positioning of the mouse and managing keyboard input, the quickest way to kill an application is to prevent the Main Loop from executing (you may have noticed that I have been stressing this).

It is extremely rare indeed that you will need to add code to the Main Loop (use a process); but, if you must, a vector has thoughtfully been provided, called *applicationMain*. This vector is normally $0000 so it won't do anything. If you wedge a routine into it be sure to end it with an RTS.

Putting it all together

Writing most GEOS applications can be done in a series of simple steps:

- Build the menu tables. Don't worry about all the service routines at first. You can point all unimplemented entries at a JMP *GotoFirstMenu*.
- Design icons and icon tables. Unimplmented icon routines can point to a RTS.
- Code the process routines if any are needed.
- Decide which vectors the application will need to use and code the routines necessary.
- Write a ColdStart routine which will initialize the screen, draw any icons, start any processes, etc. This routine should end with a RTS. From that point on the Main Loop will control the application.

Figures 1 and 2 list the pseudo-code for the Interrupt and the Main Loop respectively. Studying these should provide you with an excellent foundation for writing your own GEOS applications.

Figure 1 - *Interrupt Level Pseudo-code*

```
InterruptLevel

; First the state of the machine is saved. This
; includes A, X, Y, and S plus r0-r15 and the
; memory configuration
;
     jsr     SaveState
;
; Now the I/O area is switched
; in. Geos 128 will also ensure that
; bank 1 is the active bank.
;
     jsr     IOIn
;
; Now dblClickCount is decremented. This
; variable is used to tell if the user clicks
; the mouse twice in rapid succession
;
     jsr     DblClick
;
  .if Geos128
; Geos 128 services the mouse here
     jsr     MouseService
  .endif
;
; Now scan the keyboard and if a key is
; found place it in the keyboard que.
;
     jsr     Keyboard
;
     jsr     Alarm    ;service alarm tone timer
;
; Normally intTopVector points
; to interruptMain.If you wedge
; a routine in here the routine
; must end with jmp InterruptMain.
;
     lda    #<intTopVector
     ldx    #>intTopVector
     jsr    CallRoutine   ;execute interruptMain
;
; Normally intBotVector is empty, i.e. $0000.
; A routine wedged in here should end with rts
;
     lda    #<intBotVector
     ldx    #>intBotVector
     jsr    CallRoutine   ;normally unused.
;
     jsr    RestoreState  ;back the way it was.
     rti
; InterruptMain is called during
; each interrupt via intTopVector.
; This routine performs the bulk of the
; interrupt's work and must be called or
; things will freeze up.
;
InterruptMain

  .if Geos64
; Geos 64 services the mouse here
     jsr     MouseService
  .endif
;
     jsr  UpdateProcesses   ;update process
                            ;timers
     jsr  UpdateSleeps      ;update sleep timers
     jsr  UpdatePrompt      ;flash text prompt
     jsr  ServiceRandom     ;get a new random
                            ;number
     rts
```

Figure 2 - *Main Loop Pseudo-code*

```
MainLoop:
; first we check the keyboard and load
; keyData. Also check if
; inputVector, mouseVector, keyVector, or
; mouseFaultVector should be called. Indirectly
; menu, icon, or otherPressVector
; will be called.
; Geos 128 will handle soft (80-col)
; sprites here.
;
     jsr     KeyboardService
;
; now we check if any processes or
; sleeping routines should be
; executed.
;
     jsr     ProcessService
     jsr     SleepService
;
; next update the time and alarm
; variables. If it is time for the
; alarm to sound call alarmTmtVector
;
     jsr     TimeService
;
; applicationMain is normally $0000. You can
; wedge your own Main Loop routines in here
;
     lda     #<applicationMain
     ldx     #>applicationMain
     jsr     CallRoutine
;
     jmp     MainLoop          ;forever
```

1541/1571 DOS M-R Command Error

A caveat for multiple byte reads

by Anton Treuenfels

One of the facilities provided by the DOS of the 1541 and 1571 drives is the memory read (M-R) command, which allows the user to examine any memory location in the drive. The same purpose is served in the computer's memory space by BASIC's PEEK function: memory read can be thought of as a PEEK function for the drive.

A difference between the two is that, while BASIC's PEEK is limited to examining one memory location at a time, a single M-R command can return the contents of up to 255 memory locations. There are two forms of the M-R command: a single-byte version and a multiple-byte version. Commodore's documentation of the M-R command in the *1541 User's Guide* mentions only the single-byte version, but says that if more than one byte is read then successive bytes come from successive memory locations. In the *1571 User's Guide,* however, Commodore says that, in the multiple-byte version of the M-R command, returned bytes come from successive memory locations.

The documentation appears to be incomplete. The single-byte version of the M-R command never returns values from more than one memory location, and the multiple-byte version becomes confused near page boundaries. If the starting address of the memory read plus the number of bytes to read are such that a page boundary must be crossed to complete the request, the bytes returned after reaching the boundary come from somewhere else altogether.

The accompanying program demonstrates the error in the multiple-byte version of the M-R command by requesting reads of the same locations in drive memory using both the multiple- and single-byte versions, then comparing the results. If the sum of the starting address and the number of bytes does not cross a page boundary, then the results match (at least when reading ROM locations). If a page boundary is crossed, the results do not match.

The error manifests itself differently depending on whether or not the starting address was the last byte of a page (of the form $xxFF). If so, then the second and succeeding bytes come from $xx00, $xx01, and so on (in other words, a page wrap). For any other starting address DOS stops returning memory bytes at the page boundary and starts returning the bytes of the **00,OK,00,00** status message over and over until the number of bytes requested is reached.

A slight variation of the multiple-byte M-R subroutine demonstrates what happens when multiple bytes are requested after a single-byte M-R command is sent. Elimination of the third parameter byte (NB$) from the PRINT# statement sends a single-byte command, then enters a loop requesting multiple return bytes. The first byte is correct and all successive bytes come from the **00,OK,00,00** status message.

The error demonstration program has been used to test a 1541 and a new ROM 1571 with identical results.

It is interesting to note that two different editions of the *1541 User's Guide* do not mention the multiple-byte version of the memory read command, although both allude to an ability to read multiple bytes. (Both editions also have an example program showing how to read multiple bytes with the single-byte version, but only one will work correctly.) The *1571 User's Guide* acknowledges the existence of the multiple-byte version but not its limitations. Possibly this was a response to third party discussions of the multiple-byte version in several books about the 1541.

Unfortunately, none of the books seem to be aware of the problems either. One can speculate that perhaps the original decision not to document the multiple-byte version was made by the persons responsible for implementing it in the first place. The ability may have been used in the early development of the DOS by persons who were aware of its limitations but found it adequate for their needs.

References

Anonymous, *Commodore 1541 Disk Drive User's Guide*, Commodore Business Machines Electronics, Ltd., September 1982

Anonymous, *VIC-1541 Single Drive Floppy Disk User's Manual, Second Edition*, Commodore Business Machines, Inc., December 1982

Anonymous, *Commodore 1571 Disk Drive User's Guide, Second Edition*, Commodore Electronics Limited, August 1985

Englisch, L. and Szczepanowski, N., *The Anatomy of the 1541*, Abacus Software, Inc., 1984

Immers, R., and Neufeld, G., *Inside Commodore DOS*, Datamost, Inc., 1984

"m-r.error.bas" - *the limitation demonstrated*

```
CI 100 print"{down} 1541/1571 dos 'm-r' command"
LP 105 print" error demonstration"
CO 110 :
HG 150 nl$= "": zr$= chr$(0)
GJ 155 def fnhb(x)= int(x/256): def fnlb(x)= x-int(x/256)*256
EB 160 :
JG 200 print "{down} start address of memory read"
BL 205 input " hex# or <q> to quit";a$
DF 210 if a$="q" then end
LC 215 gosub 505: sa= a: if sa>65535 then 200
JE 220 input "{down} hex# of bytes to read";a$
FI 225 gosub 505: nb= a: if nb<1 or nb>255 then 220
HF 230 if fnhb(sa)= fnhb(sa+nb-1) then print"{down} match expected>";: goto 240
LJ 235 print"{down} mismatch expected>";
IJ 240 gosub 305
AK 245 gosub 405
MK 250 if mb$=sb$ then print " match found": goto 260
EI 255 print " mismatch found"
EB 260 for i=1 to nb
PB 265 print fnhb(sa+i-1),asc(mid$(sb$,i,1)),asc(mid$(mb$,i,1))
CB 270 next
LK 275 goto 200
MI 280 :
FB 300 rem *multi-byte read
DO 305 mb$= nl$
EP 310 open 15,8,15
PP 315 lb$= chr$(fnlb(sa)): hb$= chr$(fnhb(sa)): nb$= chr$(nb)
BL 320 print# 15,"m-r";lb$;hb$;nb$
FF 325 for i=1 to nb
FM 330 get# 15,a$: if a$=nl$ then a$= zr$
JH 335 mb$= mb$+a$
IF 340 next
AA 345 close 15
KH 350 return
HN 355 :
MN 400 rem *single-byte read
DF 405 sb$= nl$
IF 410 open 15,8,15
PK 415 for i=1 to nb
FK 420 lb$= chr$(fnlb(sa+i-1)): hb$= chr$(fnhb(sa+i-1))
HK 425 print# 15,"m-r";lb$;hb$
JC 430 get #15, a$: if a$=nl$ then a$= zr$
FP 435 sb$= sb$+a$
ML 440 next
EG 445 close15
ON 450 return
LD 455 :
LL 500 rem *hex->dec
PN 505 a=0: for i=1 to len(a$): b= asc(mid$(a$,i,1))-48: a= a*16+b+7*(b>9): next
KB 510 return
```

C128 Simple Disk Monitor

Extending the built-in monitor

by Anton Treuenfels

There are times (often shortly after scratching a file I really didn't mean to) when I find it very useful to have a disk monitor program handy. Using one of these widely available programs I can examine and, if I wish, modify the contents of any disk sector. On the other hand, I found that many of these programs were big, clumsy, rigid, and inconvenient to use. The program presented here was designed to overcome these perceived problems. Although it perhaps does not do everything that could possibly be wished for, it is relatively small, nimble, flexible, and easy to use.

A disk monitor program must at least be able to read, display, edit and write disk sectors. The problems of display and editing are shared by monitor programs in general, and *Diskmon128* deals with them by wedging itself into the main command loop of the C128's built-in machine language monitor. This reduces display and editing to problems that have already been solved (always a useful programming technique).

As bonuses *Diskmon128* gains use of the @ disk wedge command and the ability to examine and modify sectors by disassembly and assembly as well as by simple memory dump. After employing so much of the power of the built-in monitor, about all that is left for *Diskmon128* to concern itself with is the proper reading and writing of disk sectors to and from the C128's memory.

Using the program

Diskmon128 may be loaded and installed from either BASIC 7.0 or the built-in monitor. From BASIC: **bload "diskmon128", sys dec("1300"), monitor**. From the monitor: **l "diskmon128", j 1300**.

Once installed, the program functions as an extension of the C128's built-in monitor. There are four new commands (in addition to all the normal ones):

```
/R [<track> <sector>]  - read sector
/W [<track> <sector>]  - write sector
/# <device> [<drive>]  - set device and drive numbers
/Q                     - disable disk monitor wedge
```

All parameters are numeric and (since a built-in monitor ROM routine is used to collect them) may be specified in any convenient base (hexadecimal, decimal, octal, or binary). Square brackets indicate optional parameters.

The read command (**/R**) copies a disk sector into a buffer in the C128's memory. Once in the C128's memory, the contents of the sector may be displayed in hexadecimal and ASCII form by using the built-in monitor's memory dump command (**m**). The buffer is located at $B00 in RAM bank 0, so the command to display the entire sector is **m b00 bff** ($B00 is the autoboot disk sector buffer. Note that this page of memory is also used for the cassette buffer, and so is incompatible with any routines which might want to reside there). If desired, the memory display may, of course, be edited in the normal manner. Alterations made in this way affect only the copy of the sector in the C128's memory however, and no changes are made to any actual disk sector until the contents of the disk sector buffer are deliberately copied back to disk using the write command (**/W**).

Both the read and write commands may optionally be followed by disk track and sector numbers. If a track and sector are specified they are checked only to see if they are each in the range 0-99, which allows the program to create a syntactically legal direct access command. It is left up to a drive's DOS to complain if the command cannot be complied with (usually because the DOS does not recognize the existence of the requested track and sector). This approach is designed to avoid having to hardcode into the program the internal arrangement of any past, present or future disk format by taking advantage of the user's knowledge and the DOS' intelligence.

If a track and sector are not specified, both the read and write commands use default track and sector values. In the case of the read command the contents of the first two bytes of the disk sector buffer are taken to be the track and sector to read. This is based on two assumptions: that the contents of the buffer represent one sector in a series of sectors logically linked together in a single file, and that the first two bytes in the buffer represent the link to the track and sector of the next sector of the file. These assumptions are often true, making it possible to easily trace forward through the system of links tying files together under Commodore DOS. Particular conditions under which the assumptions are untrue include reaching the last sector of a file and before the program has read its first sector.

The write command defaults to using the values found in the current track and current sector variables maintained by the program. Since these variables will usually have last been set by the most recent read command, the default action normally amounts to putting the (possibly modified) sector currently in the disk sector buffer back where it came from.

The /# command is used to set the device and drive to which the program will read and write. The program defaults to device 8, drive 0. The /**Q** command resets the built-in monitor's command indirect vector to the value it had when the disk monitor was first installed, which effectively disables the disk monitor.

About the program

The main command loop of the built-in monitor is designed to accept a line of input, find the first non-space character on the line, and then jump through an indirect vector. Normally this vector points to a routine which tries to match that first character to the commands the monitor knows. *Diskmon128* re-points the indirect vector to its own match routine, which checks if the character found is the disk monitor wedge character (/). If not, control passes to the original command match routine.

If it is the wedge character, the program attempts to match the second non-space character of the input line to a known disk monitor command, and reports an error if it cannot do so (two-character rather than one-character commands are required mainly because the letter R is already employed by the built-in monitor for its Register command and no alternative one-character command seemed to make as much intuitive sense as /**R**ead).

Diskmon128 opens and closes a direct access channel to a drive for every read or write attempt rather than opening and closing once for each session with a disk. The time cost of doing this is virtually unnoticeable, and it actually saves code space since session management commands (eg., changing disks) are not needed.

High-level Kernel file routines are used exclusively rather than going to the low-level Kernel serial bus routines. There are a number of mass-storage devices which patch themselves into the indirect vectors of the high-level routines, and the program should work with any of them which recognize the normal Commodore DOS direct access commands (eg., an SFD-1001 drive with a parallel cable should, but an REU with a Commodore RAMDOS program won't). *[RAMDOS does not implement a track and sector method of data storage. - Ed.]*

Getting in and getting out

It is unfortunate that after having provided a documented method of intercepting the built-in monitor, Commodore did not provide a documented method of returning to it. The point of interception is the command dispatch routine. In the built-in monitor this routine is at the same level as the main loop, so calls from and returns to the main loop are handled as direct jumps (as opposed to BASIC 7.0's command dispatch routine, which is a subroutine of the main loop. After being intercepted, control can be returned to the main loop simply by executing an RTS instruction).

This is a workable, if not exactly academically sanctioned, method of doing things. However it would be handier if Commodore had provided another entry in the jump table at the start of the built-in monitor's ROM; one that pointed to the start of the main command loop. This would make returning after interception less of a risky business. As it is, *Diskmon128* is vulnerable to a ROM revision (in for a penny, in for a pound - the decision to use a ROM routine to collect numeric parameters must be blamed solely on a desire to save about 80 bytes of code. It is just as vulnerable to ROM revisions, and there is really no excuse for it. I simply yielded to temptation on this one "as long as I have to use an undocumented return call anyway...").

Observations and possibilities

A couple of final observations. The first is that *Diskmon128* is less than 700 bytes long, and the built-in monitor's 4K ROM contains over 1100 unused bytes. A possible use of this empty space immediately suggests itself (at least to me); although this might be of more interest to hobbyists than to Commodore itself.... The possibility is left open by making sure that, aside from the program code and disk sector buffer, all RAM usage is confined to areas already utilized by the built-in monitor (mostly variables at \$60-\$68 and \$A80-\$ABF, but also the input buffer at \$200 and the stack pointer register save at \$09).

The other observation is that the C128's Kernel contains a limited direct track and sector read ability in the BOOT CALL routine at \$FF53. This routine can read track 1, sector 0 off a disk in any drive into the autoboot sector buffer at \$B00. It wouldn't take many of the over 600 unused bytes in the Kernel ROM patch area to add a routine capable of reading and writing any track and sector on a disk in any drive, along with a jump table entry to the routine. Although the utility of such a routine to anything other than a disk monitor program might be questionable, it would offer a slightly higher-level approach to direct track and sector reads and writes than requiring such a monitor to be aware of the messy details itself.

Listing 1: *Merlin source for Diskmon128*

```
* c128 simple disk monitor
* last revision:  09/23/88

* written by anton treuenfels
* 5248 horizon drive
* fridley, minnesota usa  55421
* 612/572-8229

* program constants
```

```
asmadr = $1300      ;assembly address

cr     = $0d
csr    = $1d
spc    = $20
que    = $3f

daclf  = 13         ;direct access channel file#
cmdlf  = 15         ;command channel file#

bnk15  = %00000000 ;bank 15

* program memory use

 dum $60            ;(monitor memory)
addr ds 2           ;pointer
 dend

reawri = $0a98      ;read/write flag

 dum $0aba          ;(monitor memory)
oldmon ds 2         ;old command vector
crrdvc ds 1         ;current device
crrdrv ds 1         ;current drive
crrtrk ds 1         ;current track
crrsct ds 1         ;current sector
 dend

dskbuf = $0b00      ;disk buffer (boot block buffer)

* monitor memory use

stkptr = $09        ;stack pointer save
accl   = $60        ;numeric accumulator
monptr = $7a        ;input buffer pointer

buf    = $0200      ;input buffer

imon   = $032e      ;command execute vector

msgbuf = $0a80      ;disk command buffer

* monitor rom

mnloop = $b08b      ;monitor main loop
numprm = $b7ce      ;get numeric parameter

* kernel vectors

clsall = $ff4a      ;close all files on device
setbnk = $ff68      ;set i/o bank
primm  = $ff7d      ;print immediate

setlfs = $ffba      ;set file#, device, command
setnam = $ffbd      ;set filename
open   = $ffc0      ;open file
chkin  = $ffc6      ;set input file
chkout = $ffc9      ;set output file
clrchn = $ffcc      ;set default i/o files
chrout = $ffd2      ;output byte
getin  = $ffe4      ;input byte

* hardware registers

mmucr = $ff00       ;memory configuration

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

 org asmadr

* install

instal lda mmucr
       pha
       lda #bnk15
       sta mmucr
       ldx #3-1
       lda #$00
]bbl   sta crrdrv,x  ;current drive, track, sector
       sta dskbuf,x  ;for /r or /w without params
       dex
       bpl ]bbl
       lda #8
       sta crrdvc
       clc
       jsr setvct    ;set wedge vector
       jsr primm
       dfb cr,cr
       txt 'diskmon128 v092388'
       dfb cr
       txt 'by anton treuenfels'
       dfb cr,00
       pla
       sta mmucr
       rts

* look for disk monitor command

dskmon cmp #'/'      ;wedge token?
       beq montok    ;b:yes
       jmp (oldmon)  ;to normal handler

montok jsr getspc    ;look for monitor command
       beq reterr
       ldx #monadr-moncmd-1
]bbl   cmp moncmd,x
       beq havcmd    ;b:found command
       dex
       bpl ]bbl
reterr ldx stkptr    ;restore stack
       txs
       jsr primm     ;report problem
       dfb csr,que,$00
retmon jmp mnloop    ;back to main loop

* found command

havcmd txa
       asl
       tax
       lda monadr+1,x
       pha
       lda monadr,x
       pha
       rts

* monitor commands

moncmd txt 'qrw#'

monadr da  exquit-1
       da  exread-1
       da  exwrit-1
       da  exdevc-1

* execute quit

exquit sec
       jsr setvct
       jmp retmon
```

```
* set indirect vector(s)

setvct ldx #2-1
]bb1   lda oldmon,x
       bcs :1       ;b:replace old vector
       lda imon,x   ;save old vector
       sta oldmon,x
       lda rplvct,x ;set new vector
:1     sta imon,x
       dex
       bpl ]bb1
       rts

rplvct da  dskmon

* execute read/write

exread lda #'1'     ;'u1'
       dfb $2c
exwrit lda #'2'     ;'u2'
       sta reawri   ;read/write flag
       jsr trksct   ;get track/sector
       jsr makdac   ;make command
       jsr clsfil   ;close files on device
       jsr opndir   ;open direct access file
       bcs :1       ;b:can't open
       jsr rwsect   ;read/write sector
       jsr clsfil
:1     jmp retmon

* open direct access disk file

opndir lda #$00
       jsr setnam
       lda #cmdlf   ;command file
       jsr opnfil
       bcs :1       ;b:error
       ldx #<dacnam
       ldy #>dacnam
       lda #1
       jsr setnam
       lda #daclf   ;direct access file
       jsr opnfil
:1     rts

dacnam txt '#'

* open disk file

opnfil tay          ;secondary address
       ldx crrdvc
       jsr setlfs
       lda #$00
       tax
       jsr setbnk
       jsr open
       bcs :1
       jsr diskst   ;check status message
       bcc :1       ;b:ok
       jsr clsfil   ;close files
       sec
:1     rts

* close disk files

clsfil lda crrdvc
       jsr clsall   ;close everything on device
       rts

* read/write sector

rwsect lda reawri
       cmp #'2'     ;write?
       beq :2       ;b:yes
       jsr sndbuf   ;command read
       bcs :1       ;b:error
       jsr rddbuf   ;read buffer
:1     rts

:2     jsr wrdbuf   ;write buffer
       bcs :3
       jsr sndbuf   ;command write
:3     rts

* copy disk buffer to computer

rddbuf ldx #daclf
       jsr chkin    ;'talk'
       bcs :1
       ldy #0
]bb1   jsr getin    ;input byte
       sta dskbuf,y
       iny
       bne ]bb1
       jsr clrchn   ;'untalk'
       clc
:1     rts

* copy computer to disk buffer

wrdbuf ldx #<dcptr0
       ldy #>dcptr0
       jsr dskcmd   ;buffer pointer to start
       bcs :1
       ldx #daclf
       jsr chkout   ;'listen'
       ldy #0
]bb1   lda dskbuf,y
       jsr chrout   ;output byte
       iny
       bne ]bb1
       jsr clrchn   ;'unlisten'
       clc
:1     rts

* make direct access command

makdac ldx #-1
]bb1   inx
       lda dcread,x ;copy command template
       sta msgbuf,x
       bne ]bb1
       lda reawri
       sta msgbuf+1 ;set command
       lda crrdrv
       ora #$30
       sta msgbuf+6 ;set drive
       lda crrtrk
       jsr hexdec
       stx msgbuf+8 ;set track
       sta msgbuf+9
       lda crrsct
       jsr hexdec
       stx msgbuf+11 ;set sector
       sta msgbuf+12
       rts

* send command in the message buffer

sndbuf ldx #<msgbuf
       ldy #>msgbuf
```

```
* send disk command with error check

sndcmd jsr dskcmd
       bcs :1
       jsr diskst
:1     rts

* send disk command

dskcmd stx addr
       sty addr+1
       ldx #cmdlf
       jsr chkout    ;'listen'
       bcs :1        ;b:error
       ldy #0
       lda (addr),y  ;first char
]bb1   jsr chrout
       iny
       lda (addr),y
       bne ]bb1
       jsr clrchn    ;'unlisten'
       clc
:1     rts

* disk command messages

dcread txt 'u1:13,0,01,00',00
dcptr0 txt 'b-p:13,0',00

* check disk status

diskst ldx #cmdlf
       jsr chkin     ;'talk'
       bcs :2        ;b:error
       jsr getin     ;first byte of status message
       cmp #'2'      ;is this an error message?
       bcc :1        ;b:no
       jsr primm
       dfb cr,$00
]bb1   jsr chrout    ;display error message
       jsr getin
       cmp #cr
       bne ]bb1      ;sets carry when true
:1     php
       jsr clrchn    ;'untalk'
       plp
:2     rts

* execute device#

exdevc jsr getbyt    ;get device#
       bcs numerr    ;b:not found
       cmp #4        ;check serial bus device#
       bcc numerr
       cmp #30+1     ;31 is bad number
       bcs numerr
       sta crrdvc
       jsr getbyt    ;get drive#
       bcs :1        ;b:not found
       cmp #1+1      ;0 or 1
       bcs numerr
       sta crrdrv
:1     jmp retmon

* convert byte to ascii decimal

hexdec cmp #99+1     ;works for 0-99
       bcs numerr    ;b:too big
       ldx #'0'-1
       sec
]bb1   inx
       sbc #10
       bcs ]bb1
       adc #'0'+10
       rts

* get track and sector

trksct jsr getbyt    ;get track#
       bcs tks1      ;b:not found
       sta crrtrk
       jsr getbyt    ;get sector#
       bcc tks2      ;b:found
numerr jmp reterr

tks1   lda reawri
       cmp #'2'      ;write?
       beq tks3      ;b:yes - use current values
       lda dskbuf    ;follow link to next sector
       sta crrtrk
       lda dskbuf+1
tks2   sta crrsct
tks3   rts

* get byte value

getbyt jsr getnum    ;get number
       bcs :1        ;b:no number
       lda acc1+2
       bne numerr    ;b:too big
       lda acc1+1
       bne numerr
       lda acc1
:1     rts

* get numeric value

getnum jsr numprm    ;get numeric parameter
       bcs numerr    ;b:too big
       bne :1        ;b:found number
       sec           ;flag not found
:1     php           ;save flag
       jsr gotdlm    ;check last char
       bne numerr    ;b:not legal terminator
       plp
       rts

* character fetches

gotdlm dec monptr
getdlm jsr getchr
       beq :1        ;b:end of input
       cmp #spc      ;check for field separators
       beq :1
       cmp #','
:1     rts

getspc jsr getchr
       beq :1        ;b:end of input
       cmp #spc
       beq getspc    ;b:eat spaces
:1     rts

getchr ldx monptr
       lda buf,x
       beq :1        ;b:end of line
       cmp #':'      ;check for other terminators
       beq :1
       cmp #que
       beq :1
       inc monptr    ;next char
:1     rts
```

Listing 2: *BASIC generator for "diskmon128.o"*

```
JP  100 rem  generator for "diskmon128.o"
NI  110 n$="diskmon128.o": rem  name of program
OI  120 nd=648: sa=4864: ch=65574

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

MF  1000 data 173,   0, 255,  72, 169,   0, 141,   0
HP  1010 data 255, 162,   2, 169,   0, 157, 189,  10
KB  1020 data 157,   0,  11, 202,  16, 247, 169,   8
DO  1030 data 141, 188,  10,  24,  32, 146,  19,  32
JG  1040 data 125, 255,  13,  13,  68,  73,  83,  75
KJ  1050 data  77,  79,  78,  49,  50,  56,  32,  86
PI  1060 data  48,  57,  50,  51,  56,  56,  13,  66
OL  1070 data  89,  32,  65,  78,  84,  79,  78,  32
FN  1080 data  84,  82,  69,  85,  69,  78,  70,  69
IN  1090 data  76,  83,  13,   0, 104, 141,   0, 255
OD  1100 data  96, 201,  47, 240,   3, 108, 186,  10
FB  1110 data  32, 108,  21, 240,  10, 162,   3, 221
PN  1120 data 127,  19, 240,  15, 202,  16, 248, 166
PO  1130 data   9, 154,  32, 125, 255,  29,  63,   0
KO  1140 data  76, 139, 176, 138,  10, 170, 189, 132
AP  1150 data  19,  72, 189, 131,  19,  72,  96,  81
NL  1160 data  82,  87,  35, 138,  19, 170,  19, 173
LP  1170 data  19, 235,  20,  56,  32, 146,  19,  76
DN  1180 data 112,  19, 162,   1, 189, 186,  10, 176
EC  1190 data   9, 189,  46,   3, 157, 186,  10, 189
PO  1200 data 169,  19, 157,  46,   3, 202,  16, 236
AE  1210 data  96,  81,  19, 169,  49,  44, 169,  50
HO  1220 data 141, 152,  10,  32,  26,  21,  32,  87
AC  1230 data  20,  32,   2,  20,  32, 202,  19, 176
FF  1240 data   6,  32,   9,  20,  32,   2,  20,  76
JF  1250 data 112,  19, 169,   0,  32, 189, 255, 169
HP  1260 data  15,  32, 230,  19, 176,  14, 162, 229
BI  1270 data 160,  19, 169,   1,  32, 189, 255, 169
OC  1280 data  13,  32, 230,  19,  96,  35, 168, 174
LD  1290 data 188,  10,  32, 186, 255, 169,   0, 170
CE  1300 data  32, 104, 255,  32, 192, 255, 176,   9
HB  1310 data  32, 201,  20, 144,   4,  32,   2,  20
JF  1320 data  56,  96, 173, 188,  10,  32,  74, 255
PC  1330 data  96, 173, 152,  10, 201,  50, 240,   9
DM  1340 data  32, 137,  20, 176,   3,  32,  34,  20
EO  1350 data  96,  32,  57,  20, 176,   3,  32, 137
DJ  1360 data  20,  96, 162,  13,  32, 198, 255, 176
EI  1370 data  15, 160,   0,  32, 228, 255, 153,   0
EO  1380 data  11, 200, 208, 247,  32, 204, 255,  24
AK  1390 data  96, 162, 192, 160,  20,  32, 150,  20
ON  1400 data 176,  20, 162,  13,  32, 201, 255, 160
AF  1410 data   0, 185,   0,  11,  32, 210, 255, 200
EC  1420 data 208, 247,  32, 204, 255,  24,  96, 162
DA  1430 data 255, 232, 189, 178,  20, 157, 128,  10
DN  1440 data 208, 247, 173, 152,  10, 141, 129,  10
NK  1450 data 173, 189,  10,   9,  48, 141, 134,  10
IM  1460 data 173, 190,  10,  32,  11,  21, 142, 136
HL  1470 data  10, 141, 137,  10, 173, 191,  10,  32
MI  1480 data  11,  21, 142, 139,  10, 141, 140,  10
FA  1490 data  96, 162, 128, 160,  10,  32, 150,  20
LC  1500 data 176,   3,  32, 201,  20,  96, 134,  96
MH  1510 data 132,  97, 162,  15,  32, 201, 255, 176
HA  1520 data  16, 160,   0, 177,  96,  32, 210, 255
MF  1530 data 200, 177,  96, 208, 248,  32, 204, 255
EH  1540 data  24,  96,  85,  49,  58,  49,  51,  44
JD  1550 data  48,  44,  48,  49,  44,  48,  48,   0
MH  1560 data  66,  45,  80,  58,  49,  51,  44,  48
NC  1570 data   0, 162,  15,  32, 198, 255, 176,  27
IF  1580 data  32, 228, 255, 201,  50, 144,  15,  32
KB  1590 data 125, 255,  13,   0,  32, 210, 255,  32
LI  1600 data 228, 255, 201,  13, 208, 246,   8,  32
HF  1610 data 204, 255,  40,  96,  32,  62,  21, 176
ME  1620 data  54, 201,   4, 144,  50, 201,  31, 176
MI  1630 data  46, 141, 188,  10,  32,  62,  21, 176
OK  1640 data   7, 201,   2, 176,  34, 141, 189,  10
KN  1650 data  76, 112,  19, 201, 100, 176,  24, 162
CO  1660 data  47,  56, 232, 233,  10, 176, 251, 105
KN  1670 data  58,  96,  32,  62,  21, 176,  11, 141
BN  1680 data 190,  10,  32,  62,  21, 144,  19,  76
BA  1690 data 103,  19, 173, 152,  10, 201,  50, 240
OJ  1700 data  12, 173,   0,  11, 141, 190,  10, 173
NB  1710 data   1,  11, 141, 191,  10,  96,  32,  78
AE  1720 data  21, 176,  10, 165,  98, 208, 224, 165
ED  1730 data  97, 208, 220, 165,  96,  96,  32, 206
HA  1740 data 183, 176, 212, 208,   1,  56,   8,  32
NA  1750 data  94,  21, 208, 203,  40,  96, 198, 122
PK  1760 data  32, 118,  21, 240,   6, 201,  32, 240
MP  1770 data   2, 201,  44,  96,  32, 118,  21, 240
MP  1780 data   4, 201,  32, 240, 247,  96, 166, 122
PA  1790 data 189,   0,   2, 240,  10, 201,  58, 240
CB  1800 data   6, 201,  63, 240,   2, 230, 122,  96
```

HCD65 Assembler Macros

Making use of assembler pseudo-ops

by Robert Rockefeller

Commodore recently published their new C128 HCD65 macro assembler. Since the HCD65 manual explains the basic function of HCD65 pseudo-ops but does not demonstrate how to use them, it seems possible that some potential users of this assembler may not be able to make full use of the macro capability. This article illustrates a few macros that the author has found useful.

Macros are created with the *.MACRO* pseudo-op and terminated with the *.ENDM* pseudo-op. Nothing new there. The **MoveW** macro is an example of a simple macro which transfers 16 bits from one variable to another.

The **LoadW** macro is more complicated in that it has two behaviours. **LoadW var1,var2** is equivalent to the simple **MoveW** macro. With **LoadW var1,CONSTANT,#** the presence of the # character as the third parameter causes **var1** to be loaded with an immediate value. HCD65's ability to test for a blank field with the *.ifnb* (if not blank) pseudo-op makes it possible to create a macro which has different actions depending on the presence or absence of a parameter.

The **LineAdrs** macro uses the *.rept* (repeat) pseudo-op to create a table of start addresses for a text screen, color matrix, or for a bitmap. For example, assuming a graphics bitmap screen starts at address $6000, **LineAdrs $6000,320** would create a table of row start addresses.

The **Mark** macro stores up to six strings with the last character having the sign bit set. Upper case characters and some punctuation characters which normally have values ranging from $C0 to $DF are automatically converted to equivalent values ranging from $60 to $7F. An error message is printed if graphics characters (values $A0 to $BF) are encountered. The **Count** macro can store up to six strings with the length of each string stored as the first byte. Both these macros make use of the *.irp* pseudo-op to process each of the six possible strings in turn. The *.irpc* pseudo-op is used to process each character of each string one character at a time.

The **Float** macro stores signed values ranging from -32768 to 32767 as floating point numbers compatible with BASIC 2.0 and BASIC 7.0. It uses a *.rept* loop to normalize the mantissa and adjust the exponent. To normalize means to shift the mantissa left until the most signifigant bit is a 1.

The **DefCtrls** macro creates symbols corresponding to the values of the CONTROL keys.

"macros.src" - *for Commodore's HCD65 assembler*

```
 *= $1300
 .nclist
 .blist


var1 .wor 1
var2 .wor 2
CONSTANT = $1234
BIT15 = %1000000000000000
POSITIVE = 0
NEGATIVE = $80


;move 16 bits.

MoveW .macro %v1a,%v1b
 lda %v1b
 sta %v1a
 lda %v1b+1
 sta %v1a+1
 .endm

 MoveW var1,var2  ;sample usage.


;this macro transfers 16 bits. Accumulator is used.

LoadW .macro %v1a,%v1b,%v1c

 .ifnb <%v1c>  ;if 3rd argument is present
 .ife '%v1c'-'#'  ;  if 3rd argument equals '#'
 lda #<%v1b  ;  then load immediate value.
 sta %v1a
 lda #>%v1b
 sta %v1a+1
 .else   ;  else print error message.
 .mssg ***** bad argument in LoadW macro *****
 .endif

 .else
 lda %v1b  ;else move 16 bits.
 sta %v1a
```

```
 lda %vlb+1
 sta %vla+1
 .endif

 .endm

 LoadW var1,CONSTANT,# ;sample usage.



;this macro creates a table of row start addresses
;for the screen or color memory or a bitmap.

LineAdrs .macro %vla,%vlb
ADR = %vla

 .rept 25
 .wor ADR
ADR = ADR+%vlb
 .endr

 .endm

 LineAdrs $0400,40 ;create a sample table.



;This macro stores a string with the last character
;having the 7bit set. The angle brackets enclosing the macro
;local variables seem to be necessary to handle spaces.
;Mark can handle up to 6 arguments.

Mark .macro %vla,%vlb,%vlc,%vld,%vle,%vlf

 .irp %v2,<%vla>,<%vlb>,<%vlc>,<%vld>,<%vle>,<%vlf>

 .ifnb <%v2>  ;don't assemble null strings.

 .irpc %v3,<%v2>

 .ifge '%v3'-$80  ;if character value > 128
 .ifge '%v3'-$c0  ;  if character is upper case range $c0..$df
CHAR = '%v3'!x$a0  ;  then convert to range $60..$7f.
 .byte CHAR
 .else   ;  else print error message.
 .mssg ***** illegal graphics character in Mark argument *****
 .endif
 .else
CHAR = '%v3'   ;else assemble character value < 128.
 .byte '%v3'
 .endif

 .endr
    ;an entire string has been assembled,
 *=*-1    ;now back up PC to last character of string.
 .byte CHAR!+128  ;now set 7bit of last character.
 .endif

 .endr

 .endm

 Mark <A b>,< c D> ;sample usage.



;This macro stores a string with the count in the first byte.
;It accepts up to 6 operands.

Count .macro %vla,%vlb,%vlc,%vld,%vle,%vlf

 .irp %v2,<%vla>,<%vlb>,<%vlc>,<%vld>,<%vle>,<%vlf>
 .ifnb <%v2>  ;don't assemble null strings.
LEN = 0    ;initialize string count to 0.
 .irpc %v3,<%v2>  ;count each character with an .irpc loop.
LEN = LEN+1
 .endr

 .byte LEN,'%v2'  ;assemble string, length byte first.
 .endif
 .endr

 .endm

 Count <Text>,<more!!!> ;sample usage.



;assemble a floating point number compatible with
;BASIC 2.0 or BASIC 7.0 floating point routines.
;Float accepts up to 6 arguments.

Float .macro %vla,%vlb,%vlc,%vld,%vle,%vlf
 .irp %v2,%vla,%vlb,%vlc,%vld,%vle,%vlf
 .ifnb <%v2> ;if argument is not blank
EXP = $90  ;  EXP = correct binary exponet for 16 bit integer.
MAN = %v2  ;  MAN = 1st 2 bytes of mantissa.

 .ife MAN ;  if mantissa = 0
 .byt 0,0,0,0,0 ;  then assemble a zero.

 .else

 .iflt MAN ;  else determine correct sign and
SIGN = NEGATIVE ;    take absolute value of MAN.
MAN = -MAN
 .else
SIGN = POSITIVE
 .endif

 .rept 15 ;    now normalize mantissa while adjusting EXP.
 .ife BIT15!.MAN
MAN = MAN*2
EXP = EXP-1
 .endif
 .endr

MAN = MAN!.!nBIT15 ;    clear sign bit of mantissa and assemble number.
 .byt EXP,>MAN!+SIGN,<MAN,0,0

 .endif

 .endif
 .endr
 .endm

 Float 1,-1,-30145 ;sample usage.



;create symbols with values corresponding to the CONTROL keys.
;works with all alphabetic CONTROL characters.

DefCtrls .macro %vl
 .irpc <%v2>,<%vl>
CTRL$%v2 = '%v2'!.$1f
 .endr
 .endm

 .mlist
 DefCtrls abcz  ;sample usage.



 .end
```

Implementing A RAMdisk

*For Abacus' **Super C** On The C64*

by Kerry Gray

Super-C, the 'other' C compiler, is sold in two versions: one for the 64 and another for the 128. The system includes support for a 'RAMdisk', an area of memory that appears to the system as a disk drive, thus allowing faster access to files. The introduction of the 1764 RAM Expansion Unit has made RAMdisks possible on the 64, but since the authors of Super-C have not yet seen fit to create such a capability, it has devolved upon hackers to crowbar it in.

Installation problems

The 1764 REU is sold with RAMDOS software that emulates a Commodore disk drive. In order to make a RAMdisk simple enough for anyone to install, it is highly desirable to make use of this software rather than expand the C-shell. Simply setting up the RAMDOS before loading the C-system won't work, since the autoboot part of the C-shell program overwrites the vectors installed by RAMDOS. Therefore, to avoid excessive modification of the C-shell, RAMdisk must be installed from within the C environment. Once the RAMDOS is enabled, the memory-resident interface page and the altered system vectors must be protected from the C shell and C programs.

Using *install.c*

The *install.c* program (Listing 1) will execute Commodore's RAMDOS installation program from within the C environment and optionally copy the C development programs, *cc*, *cl* and *ce* to the RAMdisk. It also changes the top-of-memory pointer to protect the RAMDOS interface page ($CF00-$CFFF). This program can be run at any time, but be sure the 1764 is installed! Type the program in, compile it and link it with the library file *libc.l*. Use $60 for the top memory page. Have your 1764 RAMDOS installation program handy (the latest one is *ramdos64.bin4.2*). Run the program and insert the RAMDOS installation disk when prompted. If you want to copy the C development files, have your original Super-C disk handy too. These files are copied with an assembler language subroutine because the install program is destroyed when the C programs are loaded. (The assembly source is in Listing 2 - it's included only for the curious. You needn't type it in because it's included in Listing 1.) Once the transfer is complete, the program restarts the C shell and you're ready to go.

This is a no-frills program; error checking is minimal. If you like pretty colours, fancy menus or honest-to-goodness error checking, feel free to add them yourself.

The astute reader may have noticed that the program moves the top memory page down two pages to $CE rather than $CF as one might expect. This is made necessary by a bug in the C compiler: it makes a wild POKE in high memory when it starts up. Normally it lands harmlessly in an unused address in I/O space. With the memory top moved down, though, the program clobbers the RAMDOS resident program. If you don't intend to use the compiler when the RAMdisk is enabled, you may change the top page to $CF. In fact, all but the largest C programs will leave this area alone. Just to be safe, though, you should re-link your C programs (including the C programs found on your Super-C disk) with the new lower memory top.

RAMdisk entomology

The RAMdisk is, alas, not a perfect fit. The RAMDOS program differs in some subtle ways from 1541 DOS, and there are a few outright bugs to boot. The difference that concerns us here is the use of disk channel one, the SAVE channel. This channel is treated specially by the 1541 DOS. It assumes that any file opened on this channel is a file to be saved and creates a write file. The 1764 treats this save channel like any other channel; it knows whether the file is a save file because it knows when your program called SAVE rather than OPEN (they are different entry points in RAMDOS). The C Editor and C Linker both create output files by means of this save channel. Why? Simply because it saves them the trouble of appending **,w** to the file name.

When the editor saves your source file it opens the save channel using a file name such as **source.c,u**. The 1541 knows this is a write file and treats it accordingly. RAMDOS, however, treats all files not explicitly opened for writing (i.e., not opened through SAVE and not suffixed with **,w** or **,a**) as read files. If the file exists, RAMDOS will open it for reading and then object when the editor tries to write to it. If the file doesn't exist, the RAMDOS will return, "62,file not found". Fortunately, we can avoid this disaster without surgery on the C files by appending the missing letters to the file name ourselves.

To save a source file to the RAMdisk you need to append **,w** to the file name. This keeps both the editor and RAMDOS happy. The C linker engages in the same vice: it tries to SAVE your object program through OPEN. Therefore, to create an object file on the RAMdisk you should append **,p,w** to its name.

The RAMDOS program stores all files contiguously in its memory rather than imitating a random access track and sector configuration. This makes for impressively fast loading but can make trouble in programs that write to more than one file at a time. Every time you append to a file, all the files stored in higher memory must be moved up to accomodate the new data. If your program is writing to several files, it may take longer to run than if the files were on a floppy disk!

You can avoid all these problems if you use the RAMdisk as you would use your original Super C disk: it should be set up as drive ‘a’ (device 8), contain all the system files (compiler, libraries, etc.), and in general be treated as a read-only device. This setup will make program development much less aggravating, since you may now eschew disk-swapping and the grindingly slow process of compiling and linking, without the heartbreak of realizing you forgot to save your day’s work to a real disk before you shut the computer off.

The usual disclaimers...

I have tested this program on my own copy of Super-C V2 (the startup screen shows **#2.02**) and with my copy of Commodore’s RAMDOS version 4.2. Earlier versions of RAMDOS have some serious bugs that can crash your computer at any time. If you don’t have the latest version you can get it from Commodore, or it can be downloaded from some online services including Compuserve (CBMCOM LIB XX) and QuantumLink. I have no reason to believe this program will work with any other version of Super-C or RAMDOS.

Listing 1: *install.c*

```
#include "stdio.h"

file f;

int i;
char c, *m;

int program[82] =
{
  0x43a9, 0xa08d, 0xa9c0, 0xaa08, 0x20a8, 0xffba,
  0x2a9,  0x9fa2, 0xc0a0, 0xbd20, 0xa9ff, 0x2000,
  0xffd5, 0x62b0, 0xa218, 0x8a00, 0xb67d, 0xe808,
  0x6e0,  0xf8d0, 0xe6c9, 0x52d0, 0x77bd, 0x9dc0,
  0x8b5,  0xd0ca, 0x6cf7, 0x801,  0x5ad,  0xaacf,
  0x1a0,  0xba20, 0xa9ff, 0xa202, 0xa09f, 0x20c0,
  0xffbd, 0x1a9,  0x2b85, 0x8a9,  0x2c85, 0x13a6,
  0x14a4, 0x2ba9, 0xd820, 0xb0ff, 0xad21, 0xc0a0,
  0x43c9, 0x5d0,  0x45a9, 0x6f4c, 0xc9c0, 0xd045,
  0xa908, 0x8d4c, 0xc0a0, 0x54c,  0x4cc0, 0x400,
  0x384c, 0x4cc0, 0xc07e, 0xcc20, 0xa2ff, 0xe8ff,
  0x91bd, 0x20c0, 0xffd2, 0xdc9,  0xf5d0, 0x5d4c,
  0x11c0, 0x2049, 0x4143, 0x274e, 0x2054, 0x4f43,
  0x5950, 0x4320, 0xd43
};

main()

{

 f=STDIO;

 while (f==STDIO)

  {
   puts("\nInsert disk with RAMDOS");
   puts(" boot program in drive 8 \n");
   puts("Then press a key\n");
   getchar();

   f=fopen("ramdos64.bin*","r,p");

   m = (int) fgetc(f);
   m |= ((int) fgetc(f)) << 8;

   if (m!=0x6300)
       {

        puts("\nRAMDOS intallation program not found.\n");
        fclose(f);f=STDIO;

       }
     else
       {

        i = fgetf(m,0x2000,f);
        if (i == 0)
          {

           puts("\nI can't read the RAMDOS program.\n");
           fclose(f);f=STDIO;

          }
       }
  }

 fclose(f);

 do
  {
   puts("\nEnter desired device # (8-14):");
   scanf("%d",&i);
  }
 while (i<8 || i>15);

 putchar(CR);

  /* set-regs program at $6200 */

 (*(char*) 0x6200) = 0xa9;      /* lda #    */
 (*(char*) 0x6201) = (char) i;  /* device   */
 (*(char*) 0x6202) = 0xa2;      /* ldx #    */
 (*(char*) 0x6203) = 0xcf;      /* page     */
 (*(char*) 0x6204) = 0x4c;      /* jmp      */
 (*(char*) 0x6205) = 0x06;      /* <$6306   */
 (*(char*) 0x6206) = 0x63;      /* >$6306   */

 /* execute RAMDOS init program */

 call((char*) 0x6200);

 /* restore C system's NMI vector */

 nmion();
```

```
/* set new memory top */

*(char*)0x04 = (char) 0xce;

puts("\nRAMDOS installed.\n");

puts("\n\nDo you wish to install C System files? ");

do c = getchar();
  while (c != 'y' && c != 'n');

putchar(c);
putchar(CR);

if (c == 'y')
  {
    puts("\nInsert C-system disk in drive 8 and press a key.\n\n");
    cmove ( (char*) 0xc000,170,program);

    getchar();

    call( (char*) 0xc000);

    /* subroutine does not return */
  }

puts("Press a key ...");

getchar();
}
```

Listing 2: *source code for ML in install.c*

```
;
; loader for hidden
; system files
;
clrchn = $ffcc
chrout = $ffd2
load   = $ffd5
save   = $ffd8
setlfs = $ffba
setnam = $ffbd
cboot  = $0400  ;c shell
patch  = $08b6  ;in fast loader
prstrt = $0801  ;c prg start addr
txttab = $2b    ;save param
devnum = $cf05  ;inside interf pg
;
       * = $c000
;
;call entry pt
;
start  lda #'c'         ;file name
       sta name+1       ;is 'cc'
;
; read in fast loader
;
restrt lda #8           ;device
       tax
       tay
       jsr setlfs
       lda #2
       ldx #<name       ;file to copy
       ldy #>name
       jsr setnam
       lda #0
       jsr load
       bcs error
;
; patch fast loader
;
       clc
       ldx #0
       txa              ;compute checksum
loop   adc patch,x      ;to ensure
       inx              ;your version
       cpx #6           ;matches mine
       bne loop
       cmp #$e6         ;chksum
       bne error
loop2  lda npatch-1,x   ;new code to
       sta patch-1,x    ;send control
       dex              ;back to me
       bne loop2
       jmp (prstrt)     ;execute loader
;
; when it finishes, fast
; loader will jump here
;
return
ramdev lda devnum       ;ramdisk dvc
       tax
       ldy #1
       jsr setlfs
       lda #2
       ldx #<name       ;same name
       ldy #>name
       jsr setnam
       lda #<prstrt     ;prg begins
       sta txttab       ;at prstrt
       lda #>prstrt
       sta txttab+1
       ldx $13          ;fast loader leaves
       ldy $14          ;end address here
       lda #<txttab
       jsr save
       bcs error
;
next   lda name+1
       cmp #'c'         ;compiler
       bne try1
       lda #'e'
       jmp xx
try1   cmp #'e'         ;editor
       bne done
       lda #'l'         ;linker
xx     sta name+1
       jmp restrt       ;continue
done
       jmp cboot        ;reboot c shell
;
npatch jmp return       ;new code for
       jmp error        ;fast loader
;
; print error message
;
error  jsr clrchn
       ldx #$ff
loop3  inx
       lda msg,x
       jsr chrout
       cmp #$d
       bne loop3
       jmp next
msg    .byte 17, 'i can';39,'t copy '
name   .byte 'cc',$d
 .end
```

SuperNumbers III

Sticky variables for C128, C64 & VIC-20

by Richard Curcio

Having successfully modified *SuperNumbers* to run on my VIC-20, I decided to attempt a C128 conversion. If you're unfamiliar with *SuperNumbers*, it is a neat utility by John R. Bennett which appeared in *Transactor*, Volume 6, Issue 1 back in July 1985. It provides the C64 with a new class of numeric variable. These new variables, which are single letters preceded by the British pound symbol (£), have fixed locations in memory and are invulnerable to program failure, CLR, and reset. They are also faster variables since BASIC doesn't have to search through its other variables to find a particular supernumber.

The "III" in the title of this article reflects the inclusion of three versions of a new *SuperNumbers* program, for the C128, C64, and VIC-20.

C64/VIC-20

On re-examining the original source program, I noticed two nearly identical sections of code. These I combined into one subroutine in the new source listings labelled, not unexpectedly, SUBRTN. This gave me room to add some simple vector management to the initialization. When *SuperNumbers* is initialized, the contents of IERROR is stored at REALERR, then IERROR is changed to point to NEWERR. If IERROR is already pointing to NEWERR, the initialization exits. This allows *SuperNumbers* to co-exist with another error-intercepting wedge. The contents of the vector IEVAL, however, are simply replaced by NEWEVAL.

In the original article, the Editor noted that there are apparently more than 26 supernumbers (SNs). The routine will accept **£1**, **£@** and even **£<shift-a>**. Unfortunately, these extra SNs are located outside the area of memory that the machine language set aside for the 26 alphabetic SNs. If the area is unused, no problem. If, on the other hand, that area of RAM is being used by another routine, assigning a value to **£%** or **£*** could cause bad things to happen. Like a crash.

The call to the ROM routine CKALPH, near the start of SUBRTN, returns with carry set if CHRGET found a letter in the BASIC text. The BCS around JMP SYNTAX ensures that *SuperNumbers* will only accept the 26 unshifted alphabetic characters. Twenty-six new variables is plenty. Besides, there is another way to get more than 26 supernumbers. This is one reason why the new CLRRAM subroutine is seemingly more complex than necessary. Stay tuned.

My first VIC conversion was done the 'hard way'. The original BASIC loader was run with a protected area of VIC memory as the destination. I then POKEd the ROM calls and absolute addresses with the proper values. (Incidentally, VIC's location for MEMTO1 really does have the two middle digits transposed from the C64 location. That's not a typo.)

For these new conversions, I had to get around the fact that the assembler I was using, LADS64, doesn't have a WORD pseudo-op. The original routine contained a table of 26 two-byte values representing the addresses of the 26 five-byte supernumbers. I couldn't use BYT since LADS only accepts decimal and ASC, not expressions for that pseudo-op, and I didn't know what the table entries would be until the whole thing was assembled.

The table was halved to 26 single-byte values. SUBRTN adds the appropriate table entry to NUMS to find the location of the supernumber BASIC is looking for. There is a slight loss of speed using this method. However, this kept the ML, with the added vector management, error check, and longer CLRRAM from growing much longer than the original routine. In fact, the C64 and VIC versions are two bytes shorter - 163 bytes. Like the original, these routines require an additional 130 bytes immediately after the ML to hold the 26 5-byte floating-point **£** variables.

C128

For the C128, my intention was to store the new variables with the ML in BANK 15, RAM 0, and gain more speed by avoiding the bank switching the interpreter performs between program text in RAM 0, and variables in RAM 1. BASIC 7.0 would meet me only half-way on this brilliant idea, willingly returning values from RAM 0, but refusing to store values there. With much grumbling, I made the necessary changes and placed the new variables high in RAM 1 at $FF45. This area is immediately after the IRQ, NMI and reset preliminaries the system copies to all banks, isolated from normal variables and unused by the system. Knowing the storage location should have eliminated the table entry question. The lookup-and-ADC method was retained for reasons that will be explained shortly.

The source listing for C128 *SuperNumbers* is very similar to the C64/VIC version. The CLRRAM subroutine is longer because of the need to use the Kernal INDSTA routine to get at the other bank. As noted, though ASCFLT is located at $8D22, entry at $78E3 performs a needed extra step. This extra step is why, if the variable is not a supernumber, NODIGIT jumps to OLDEVAL +14 in the 128, and OLDEVAL +12 in the 64/VIC source listing.

Even with the unavoidable bank switching, C128 supernumbers are considerably faster than normal variables, especially so in FAST mode. One thing the 128 really needs is faster variables.

The Loaders

All three loaders will relocate the ML to an address of your choice by changing the variable **sn** in line 120. The VIC version POKEs the ML into RAM in Block 5, the slot for auto-start cartridges. For other VIC-20 locations, the usual top of memory lowering POKEs should be performed before running the loader. C64/VIC *SuperNumbers* requires 293 bytes for the ML and variable storage.

The loader for *SuperNumbers 128* puts 173 bytes of ML at address 4864/$1300, in the 'applications' area. The £ variable area defaults to 65439/$FF45 in RAM 1. However, all three versions can have their variables elsewhere, which brings us to the next topic.

More Than XXVI

The lookup-and-ADC method of finding the address of a £ variable had the happy side benefit of allowing the storage area to be moved with just two POKEs. By performing these POKEs on the fly, *SuperNumbers* can be made to have more than one set of variables. If **sn** equals the start address of the routine, the low byte of the storage area is contained in **sn +108**, and the high byte at **sn +110**. PEEK these locations and put the values into normal variables if you intend to restore the default storage area.

For the 128, the storage area *must* be in RAM 1. Setting aside memory in RAM 1 is simple enough. Locations 47/48 contain the pointer for the start of variables; and 57/58, the pointer for the end of strings. In direct mode, or at the start of a BASIC program, before any variables are assigned, **poke 47, 0: poke 48, 5** raises the normal start of variables by 256 bytes and **poke 57, 0: poke 58, 254** lowers the end of strings by 256 bytes. These POKEs should be followed by CLR.

When you want to switch to a different set of supernumbers, your program would first POKE **sn +108** and **sn +110** with the values corresponding to the low byte and high byte of the start of the new storage area. (Obviously, £ variables can't be used for these POKEs.) If enough memory has been set aside for the purpose, BASIC subroutines could have their own sets of supernumbers - local variables - like higher level languages. Local supernumbers could be passed to the main program (the global variables) by **A = £A**. Two POKEs just before the subroutine returns will restore the first storage area or any other set of supernumbers.

SuperNumbers' 'cold' start clears the storage area pointed to by **sn +108** and **sn +110**. If *SuperNumbers* is the only error wedge in place, or the first in a chain (the last one enabled), COLD can be called again to clear the alternate storage area. The CLRRAM routine can be called separately with **sys sn +119** on all three versions.

Finally

Thanks must go to John R. Bennett for making his program available to *Transactor* readers. Converting his original work from the C64 to the C128, making changes as I encountered obstacles along the way, provided me with an interesting project.

Listing 1: *Source.128 - LADS format*

```
1000 *= $1300                    ; c128 .org
1010 ;
1020 .d       sn128.obj
1030 .s
1040 ;
1050 ;supernumbers revisited
1060 ;
1070 ;adapted by r.curcio
1080 ;from a program by john bennett (vol. 6, iss. 01)
1090 ;
1100 ;lads format
1110 ;
1120 ;
1130 chrget  = $0380
1140 ;
1150 valtyp  = $0f
1160 intflg  = $10
1170 ;
1180 ierror  = $0300
1190 ieval   = $030a
1200 ;
1210 oldeval = $78da
1220 ;
1230 ;routine which sets carry if accumulator holds a letter
1240 ckalph  = $7b3c
1250 ;
1260 ;routine to load facl with number in raml pointed to by a,y
1270 memtol  = $7a85
1280 ;
1290 ;routine to change ascii to floating point
1300 ascflt  = $78e3
1310 ;routine is really at $8d22. entry here first performs ldx #$00
1320 ;
1330 ;
1340 syntax  = $796c
1350 indsta  = $ff77
1360 ;
1370 nums    = $ff45             ;storage for \ variables
1380 ;
```

```
1390 cold    jsr clrram        ;zero storage area
1400 ;
1410 warm    lda ierror
1420         ldy ierror+1
1430         cmp #<newerr      ;test ierror already
1440         bne chngvec       ;points to newerr
1450         cpy #>newerr
1460         beq leave
1470 chngvec sta realerr+1
1480         sty realerr+2
1490         lda #<newerr
1500         ldy #>newerr
1510         sta ierror
1520         sty ierror+1
1530         lda #<neweval
1540         ldy #>neweval
1550         sta ieval
1560         sty ieval+1
1570 leave   rts
1580 ;
1590 ;
1600 newerr  cpx #11           ;syntax"?
1610         bne realerr
1620         cmp #"\"
1630         beq found
1640 ;
1650 realerr jmp $ffff         ;not supernumber
1660 ;
1670 found   jsr subrtn
1680         ldx #0
1690         stx valtyp
1700         stx intflg
1710         rts
1720 ;
1730 neweval lda #0
1740         sta valtyp
1750         jsr chrget
1760         bcs nodigit
1770         jmp ascflt        ;not supernumber
1780 ;
1790 ;
1800 nodigit cmp #"\"
1810         beq found1
1820         jmp oldeval+14    ;not supernumber
1830 ;
1840 ;
1850 found1  jsr subrtn
1860         jmp memto1
1870 ;
1880 ;
1890 subrtn  jsr chrget        ;get character
1900         jsr ckalph        ;make sure its a letter
1910         bcs okay
1920         jmp syntax
1930 okay    sbc #"a"
1940         tax
1950         jsr chrget        ;point to next
1960 numadr  lda #<nums        ;calculate address
1970         ldy #>nums        ;of supernumber
1980         clc
1990         adc addtab,x
2000         bcc endsub
2010         iny
2020 endsub  rts
2030 ;
2040 ;
2050 clrram  ldx #0            ;set supernumbers
2060         jsr numadr        ;to zero
2070         sta $c3
2080         sty $c4
2090         lda #$c3          ;set up pointer
2100         sta $02b9
2110         ldy #$00          ;counter
2120 zero    lda #$00          ;byte to store
2130         ldx #$01          ;bank 1
2140         jsr indsta
2150         iny
2160         cpy #130
2170         bne zero
2180         rts
2190 ;
2200 ;table of values used to calculate
2210 ;address of supernumber
2220 ;
2230 addtab  .byt   0   5  10  15  20
2240         .byt  25  30  35  40  45
2250         .byt  50  55  60  65  70
2260         .byt  75  80  85  90  95
2270         .byt 100 105 110 115 120
2280         .byt 125
2290 ;
```

Listing 2: *Source.64v - LADS format*

```
1000 *= $c800                    ; c64 .org
1010 ;
1020 .d      sn64.obj
1030 .s
1040 ;
1050 ;supernumbers revisited
1060 ;
1070 ;adapted by r.curcio
1080 ;from a program by john bennett (vol. 6, iss. 01)
1090 ;
1100 ;lads format
1110 ;
1120 ;
1130 chrget  = $0073
1140 ;
1150 valtyp  = $0d
1160 intflg  = $0e
1170 ;
1180 ierror  = $0300
1190 ieval   = $030a
1200 ;
1210 oldeval = $ae86             ;$ce86 for vic
1220 ;
1230 ;routine which sets carry if accumulator holds a letter
1240 ckalph  = $b113             ;$d113 for vic
1250 ;
1260 ;routine to load fac1 with number pointed to by a,y
1270 memto1  = $bba2             ;$dab2 for vic
1280 ;
1290 ;routine to change ascii to floating point
1300 ascflt  = $bcf3             ;$dcf3 for vic
1310 ;
1320 ;
1330 ;
1340 syntax  = $af08             ;$cf08 for vic
1350 ;
1360 ;
1370 ;
1380 ;
1390 cold    jsr clrram          ;set ram to zero
1400 ;
1410 warm    lda ierror
1420         ldy ierror+1
1430         cmp #<newerr        ;test ierror already
1440         bne chngvec         ;points to newerr
1450         cpy #>newerr
1460         beq leave
1470 chngvec sta realerr+1
1480         sty realerr+2
1490         lda #<newerr
1500         ldy #>newerr
1510         sta ierror
1520         sty ierror+1
1530         lda #<neweval
1540         ldy #>neweval
```

```
1550          sta ieval
1560          sty ieval+1
1570 leave    rts
1580 ;
1590 ;
1600 newerr   cpx #11              ;syntax"?
1610          bne realerr
1620          cmp #"\"
1630          beq found
1640 ;
1650 realerr  jmp $ffff            ;not supernumber
1660 ;
1670 found    jsr subrtn
1680          ldx #0
1690          stx valtyp
1700          stx intflg
1710          rts
1720 ;
1730 neweval  lda #0
1740          sta valtyp
1750          jsr chrget
1760          bcs nodigit
1770          jmp ascflt           ;not supernumber
1780 ;
1790 ;
1800 nodigit  cmp #"\"
1810          beq found1
1820          jmp oldeval+12       ;not supernumber
1830 ;
1840 ;
1850 found1   jsr subrtn
1860          jmp memtol
1870 ;
1880 ;
1890 subrtn   jsr chrget           ;get character
1900          jsr ckalph           ;set carry if letter
1910          bcs okay
1920          jmp syntax
1930 okay     sbc #"a"
1940          tax
1950          jsr chrget           ;point to next
1960 numadr   lda #<nums           ;calculate address
1970          ldy #>nums           ;of supernumber
1980          clc
1990          adc addtab,x
2000          bcc endsub
2010          iny
2020 endsub   rts
2030 ;
2040 ;
2050 clrram   ldx #0               ;get start of storage
2060          jsr numadr
2070          sta $c3
2080          sty $c4
2090          txa
2100          tay
2110 zero     sta ($c3),y
2120          iny
2130          cpy #130
2140          bne zero
2150          rts
2160 ;
2170 ;
2180 ;
2190 ;
2200 ;table of values used to calculate
2210 ;address of supernumber
2220 ;
2230 addtab   .byt   0   5  10  15  20
2240          .byt  25  30  35  40  45
2250          .byt  50  55  60  65  70
2260          .byt  75  80  85  90  95
2270          .byt 100 105 110 115 120
2280          .byt 125
2290 ;
2300 nums     .byt 0               ;storage starts here
```

Listing 3: *C128.ldr*

```
MI  100 rem *** supernumbers loader ***
OL  110 rem *** c128 version ***
DD  120 sn=4864:bank15:rem will relocate
BO  130 ck=0
DK  140 readd:ck=ck+d:ifd=999then160
LD  150 goto140
JB  160 ifck<>15872thenprint"error in data":end
CM  170 restore
KM  180 na=sn
JI  190 readd:ifd=999then240
GB  200 ifd=>0thenpokena,d:goto230
CB  210 ad=sn+abs(d):gosub350
KA  220 pokena,lb:na=na+1:pokena,hb
CJ  230 na=na+1:goto190
DH  240 ad=sn+44:gosub350
MK  250 pokesn+10,lb:pokesn+24,lb
EL  260 pokesn+14,hb:pokesn+26,hb
KJ  270 ad=sn+65:gosub350
EN  280 pokesn+34,lb:pokesn+36,hb
AE  290 rem
KE  300 rem
PJ  310 print"supernumbers installed":printsn"to"na-1
LO  320 print"coldstart = sys"sn
JD  330 print"warmstart = sys"sn+3
EF  340 end
AI  350 hb=ad/256:lb=ad-int(ad/256)*256:return
BP  1000 data  32,-119, 173,   0,   3, 172,   1,   3
BP  1010 data 201,  44, 208,   4, 192,  19, 240,  26
HD  1020 data 141, -53, 140, -54, 169,  44, 160,  19
OO  1030 data 141,   0,   3, 140,   1,   3, 169,  65
LJ  1040 data 160,  19, 141,  10,   3, 140,  11,   3
OB  1050 data  96, 224,  11, 208,   4, 201,  92, 240
OD  1060 data   3,  76, 255, 255,  32, -90, 162,   0
GD  1070 data 134,  15, 134,  16,  96, 169,   0
IM  1080 data 133,  15,  32, 128,   3, 176,   3,  76
KG  1090 data 227, 120, 201,  92, 240,   3,  76, 232
PK  1100 data 120,  32, -90,  76, 133, 122,  32, 128
HN  1110 data   3,  32,  60, 123, 176,   3,  76, 108
EJ  1120 data 121, 233,  65, 170,  32, 128,   3, 169
MD  1130 data  69, 160, 255,  24, 125,-148, 144,   1
OO  1140 data 200,  96, 162,   0,  32,-107, 133, 195
BN  1150 data 132, 196, 169, 195, 141, 185,   2, 160
DP  1160 data   0, 169,   0, 162,   1,  32, 119, 255
NJ  1170 data 200, 192, 130, 208, 244,  96,   0,   5
EJ  1180 data  10,  15,  20,  25,  30,  35,  40,  45
OO  1190 data  50,  55,  60,  65,  70,  75,  80,  85
NH  1200 data  90,  95, 100, 105, 110, 115, 120, 125
II  1210 data 999
```

Listing 4: *C64.ldr*

```
MI  100 rem *** supernumbers loader ***
NO  110 rem *** c64 version ***
FA  120 sn=51200:rem will relocate
BO  130 ck=0
DK  140 readd:ck=ck+d:ifd=999then160
LD  150 goto140
BC  160 ifck<>15585thenprint"error in data":end
CM  170 restore
KM  180 na=sn
JI  190 readd:ifd=999then240
GB  200 ifd=>0thenpokena,d:goto230
CB  210 ad=sn+abs(d):gosub350
KA  220 pokena,lb:na=na+1:pokena,hb
CJ  230 na=na+1:goto190
DH  240 ad=sn+44:gosub350
MK  250 pokesn+10,lb:pokesn+24,lb
EL  260 pokesn+14,hb:pokesn+26,hb
KJ  270 ad=sn+65:gosub350
EN  280 pokesn+34,lb:pokesn+36,hb
```

```
GJ  290 ad=sn+164:gosub350
HO  300 pokesn+108,lb:pokesn+110,hb
PJ  310 print"supernumbers installed":printsn"to"na-1
LO  320 print"coldstart = sys"sn
JD  330 print"warmstart = sys"sn+3
EF  340 end
AI  350 hb=ad/256:lb=ad-int(ad/256)*256:return
BP  1000 data  32,-119, 173,   0,   3, 172,   1,   3
KP  1010 data 201,  44, 208,   4, 192, 200, 240,  26
ON  1020 data 141, -53, 140, -54, 169,  44, 160, 200
OO  1030 data 141,   0,   3, 140,   1,   3, 169,  65
OP  1040 data 160, 200, 141,  10,   3, 140,  11,   3
OB  1050 data  96, 224,  11, 208,   4, 201,  92, 240
OD  1060 data   3,  76, 255, 255,  32, -90, 162,   0
LB  1070 data 134,  13, 134,  14,  96, 169,   0, 133
OK  1080 data  13,  32, 115,   0, 176,   3,  76, 243
NJ  1090 data 188, 201,  92, 240,   3,  76, 146, 174
HC  1100 data  32, -90,  76, 162, 187,  32, 115,   0
IB  1110 data  32,  19, 177, 176,   3,  76,   8, 175
JJ  1120 data 233,  65, 170,  32, 115,   0, 169, 164
IG  1130 data 160, 200,  24, 125,-138, 144,   1, 200
PO  1140 data  96, 162,   0,  32,-107, 133, 195, 132
FN  1150 data 196, 138, 168, 145, 195, 200, 192, 130
HL  1160 data 208, 249,  96,   0,   5,  10,  15,  20
OL  1170 data  25,  30,  35,  40,  45,  50,  55,  60
EF  1180 data  65,  70,  75,  80,  85,  90,  95, 100
CO  1190 data 105, 110, 115, 120, 125, 999
```

Listing 5: *vic.ldr*

```
MI  100 rem *** supernumbers loader ***
II  110 rem *** vic version ***
PC  120 sn=40960:rem will relocate
BO  130 ck=0
DK  140 readd:ck=ck+d:ifd=999then160
LD  150 goto140
DA  160 ifck<>15600thenprint"error in data":end
CM  170 restore
KM  180 na=sn
JI  190 readd:ifd=999then240
GB  200 ifd=>0thenpokena,d:goto230
CB  210 ad=sn+abs(d):gosub350
KA  220 pokena,lb:na=na+1:pokena,hb
CJ  230 na=na+1:goto190
DH  240 ad=sn+44:gosub350
MK  250 pokesn+10,lb:pokesn+24,lb
EL  260 pokesn+14,hb:pokesn+26,hb
KJ  270 ad=sn+65:gosub350
EN  280 pokesn+34,lb:pokesn+36,hb
GJ  290 ad=sn+164:gosub350
HO  300 pokesn+108,lb:pokesn+110,hb
PJ  310 print"supernumbers installed":printsn"to"na-1
LO  320 print"coldstart = sys"sn
JD  330 print"warmstart = sys"sn+3
EF  340 end
AI  350 hb=ad/256:lb=ad-int(ad/256)*256:return
BP  1000 data  32,-119, 173,   0,   3, 172,   1,   3
MP  1010 data 201,  44, 208,   4, 192, 160, 240,  26
OO  1020 data 141, -53, 140, -54, 169,  44, 160, 160
OO  1030 data 141,   0,   3, 140,   1,   3, 169,  65
DB  1040 data 160, 160, 141,  10,   3, 140,  11,   3
OB  1050 data  96, 224,  11, 208,   4, 201,  92, 240
OD  1060 data   3,  76, 255, 255,  32, -90, 162,   0
LB  1070 data 134,  13, 134,  14,  96, 169,   0, 133
OK  1080 data  13,  32, 115,   0, 176,   3,  76, 243
JG  1090 data 220, 201,  92, 240,   3,  76, 146, 206
BD  1100 data  32, -90,  76, 178, 218,  32, 115,   0
CB  1110 data  32,  19, 209, 176,   3,  76,   8, 207
JJ  1120 data 233,  65, 170,  32, 115,   0, 169, 164
NH  1130 data 160, 160,  24, 125,-138, 144,   1, 200
PO  1140 data  96, 162,   0,  32,-107, 133, 195, 132
FN  1150 data 196, 138, 168, 145, 195, 200, 192, 130
HL  1160 data 208, 249,  96,   0,   5,  10,  15,  20
OL  1170 data  25,  30,  35,  40,  45,  50,  55,  60
EF  1180 data  65,  70,  75,  80,  85,  90,  95, 100
CO  1190 data 105, 110, 115, 120, 125, 999
```

Inside The 1764 REU

Half way to a one meg 64

by Paul Bosacki

This article was written in two parts. The first bit was written in early January, just after the mail started to come in on *Care and Feeding of the C256*. The rest of it was written four months later when some additional information led me to question one of my early speculations. But I've left the first part mostly as it was because I thought you might appreciate reading about the mental stretching that owning a computer has required.

Ever since *Transactor* published *Care and Feeding of the C256*, I've been receiving a little mail. Most of it has had to do with my 1 meg 64, and how this was achieved. The answer lies in the marriage of two distinct memory expansion strategies. The first is a commercial 512K RAM Expansion Unit that uses high speed, direct memory access techniques; the second, 512k of user installed, slower, bank-switched RAM along the lines of the 256K project from *Transactor*, Volume 9, Issue 2. I won't be discussing the bank switched RAM project here. That's for later. Rather, it's the 512K REU that I want to talk about. It's a 1764 REU. You know, the 256K type.

There can be little doubt that the 1764 REU is one of the most significant peripherals that Commodore ever offered for the 64. In lieu of a faster processor, it is the best thing going for speeding up an otherwise slow program. As long, that is, as the program takes advantage of the REU. GEOS is one of those programs. And no one, I imagine, would be foolish enough to suggest that an REU of some sort isn't necessary when running GEOS - and the bigger the better. Put a 256K REU on GEOS, and you're soon wishing you had 512K. The improvements an REU offers are just short of amazing.

This is the story of curiosity and the cat, including the satisfaction part. What follows is a look inside the REU and, best of all, the information necessary to expand the 1764 REU to 512K. I have heard that some of this information is available elsewhere; however, it's important enough to bear repeating.

REU Internals

Anyone who's ever been brave enough to open a 1764 has been greeted with a few very intriguing words on the printed circuit board: C128 RAM Expansion. It seems that Commodore uses one REC (RAM Expansion Controller) but populates each REU with varying quantities or types of RAM. Simply put, the 1764 is a 1750 REU with only one bank of 41256-15 DRAM and a different name stenciled on the case. In fact, the 1700 REU is different from its beefier siblings only in that it uses the 4164 DRAM rather than the 41256.

But all that's intriguing doesn't stop there. At the lower right of the PC board is an empty layout for a 28-pin chip: either an 8, 16 or 32K EPROM. Close inspection of the board reveals that the data bus and the low 13 address lines are brought out to the layout. As well, A14 can be selected by resetting a jumper on the board. Of the two EPROM select lines, one is directly connected to the REC; the other is connected to ground. Yet, how the ROM is accessed by the 64 is a partial mystery to me.

Let me outline some of my observations. First, the REC itself selects the EPROM in response to a low on either the /ROMH or /ROML line of the cartridge port. There's a fly in that ointment though. A cartridge signals the presence of an EPROM to the C64 by pulling low the /EXROM or /GAME line (or both) of the expansion port. These lines are directly connected to the PLA and default high. But, the /EXROM and /GAME lines are not manipulated by the REU. Put another way, the EPROM won't be selected because the REU hasn't indicated to the 64 that there's an EPROM present! Now this confounds me. Why the layout for an EPROM, but no way to access it?

There are two possibilities here. One, I'm missing something obvious. Or two, an early design aspect didn't make it all the way to the production stages. You see, it is possible to select the EPROM, but we have to cheat. On the REU board, connect the /EXROM line to ground. An 8K EPROM is then selected in the $8000 range. Or connect the /GAME line to ground along with /EXROM to allow a 16K EPROM. Now an EPROM would be selected from $8000 to $BFFF. However, we're always going to be out 8K of system RAM. And in the second case, we must supply BASIC, however modified (otherwise, why bother with a 16K EPROM?).

Speculations

Now, here's a piece of speculation for you. The EPROM is obviously selected by the REC based on the status of /ROML and /ROMH. But if the EPROM were always to be selected, why not connect the lines directly to the EPROM? Two possible reasons. Neglecting for a moment that A14 can be routed to the layout, the EPROM can be either an 8 or 16K chip. This way the REC handles the decoding. Or perhaps there is the option to map in the EPROM. A look at the REC registers reveals that three bits in the command register are marked reserved. Perhaps these bits serve that function. This means that an output from the REC might connect directly to the /EXROM line, another to /GAME. By setting or clearing a bit, a low on the corresponding line would signal to the 64 whether an EPROM were present. However, I couldn't get any combination of those bits to act in that fashion. Again, maybe I'm missing something.

Then there's the fact that by changing J2, A14 can be brought out to the EPROM. This allows a 32K EPROM to occupy the layout. But how is the upper 16K selected? Do we again look to the suspect bits in the command register? Is that function somehow hidden there as well?

So, what was the EPROM to contain? Your guess is as good as mine. Speculation: maybe custom REU routines. Perhaps Commodore intended a custom RAMDOS to be placed there. It's only conjecture. And as I said, it seems impossible for the REC to select the EPROM without some form of intervention on our part; i.e., connecting either /EXROM, or both /EXROM or /GAME, to ground. Then, whatever code an EPROM contained would be up to the individual. Unfortunately, it seems that the layout made it to the board but supporting C64 select functions didn't. Anybody know for certain?

We're here to pump REUs up

Intriguing point number 3: on the solder side of the circuit board is a jumper with the promising words: 512K-cut. When this line is cut, it signals to the REC that 256K bit DRAMs are present. What this suggests is that all those people out there who bought the 1700 REU and now crave 512K of expansion RAM need only unsolder the two banks of 4164s and install 41256s - and cut the jumper, of course. One bank of 41256-15's would yield 256K, another would take the REU to its maximum 512k. However, it's only fair to warn you that I have not done this, and cannot, therefore, assure you of the results. And it would be a finicky job - desoldering 256 pins! If anyone does succeed at this, let me know!

For those who own 1764s the process is simpler. Because the 512K jumper is already cut, all that needs to be done is install another bank of 41256-15s. First, carefully disassemble your 1764 (needless to say, such action will void your warranty). The case is held together by four plastic posts set in sockets. It's best to start at the expansion port connector and pry up with a small screwdriver. Work your way around the REU as the case gives. Inside is the RF shield which just pries off, then you're at the board itself. Compare the board against the drawing. If they're not significantly the same, proceed at your own risk.

However, if that is the case, all is not lost. Check a couple of things out. Look for the jumper on the solder side of the board. It's beneath the REC. If it's there, chances are that simply adding the extra RAM will work even if your board is different. And check for the C128 RAM Expansion title. I can't help but feel that's a dead giveaway.

If every thing checks out, look below the bank of 41256s labeled Bank 1. You will see the layouts for eight sixteen-pin chips; just to the right of the layouts will be the words "Bank 2" (another dead giveaway). Clear each of the solder pads using either a vacuum desolder or solder braid and install eight 41256-15s there. You'll want to observe the usual anti-static precautions. Ground yourself first! With DRAM costing $12.00 a chip, mistakes are expensive. You may want to install the DRAMs in sockets to minimize the possibility of damage. Use low profile sockets. With the RF shield back in place, there's just enough room. But check first, just to be certain.

Once the chips are installed, you're done. Put the unit back together and pull out your copy of GEOS 1.3/2.0 or the utility disk that came with your REU. The RAMdisk utility from that disk will configure all 512K (minus some room for code) as a RAMdisk. It's worth noting however, that the REU test program tests only the first 256K.

Under GEOS, your options are somewhat varied. Under 1.3, you can configure a RAM 1541 and a Shadow 1541, or if you have two 1541's, two Shadow 1541's. Under 2.0, configure a RAM 1571 or whatever. If you want to test the additional RAM, load up a RAM 1571 and click on **validate**. If your RAMdisk validates then your new RAM passes. It's not a complete test: only track and sector links are being fetched, so you're really only testing the first two bytes of each RAM page. However, in most cases, that test alone will tell you if there's a problem.

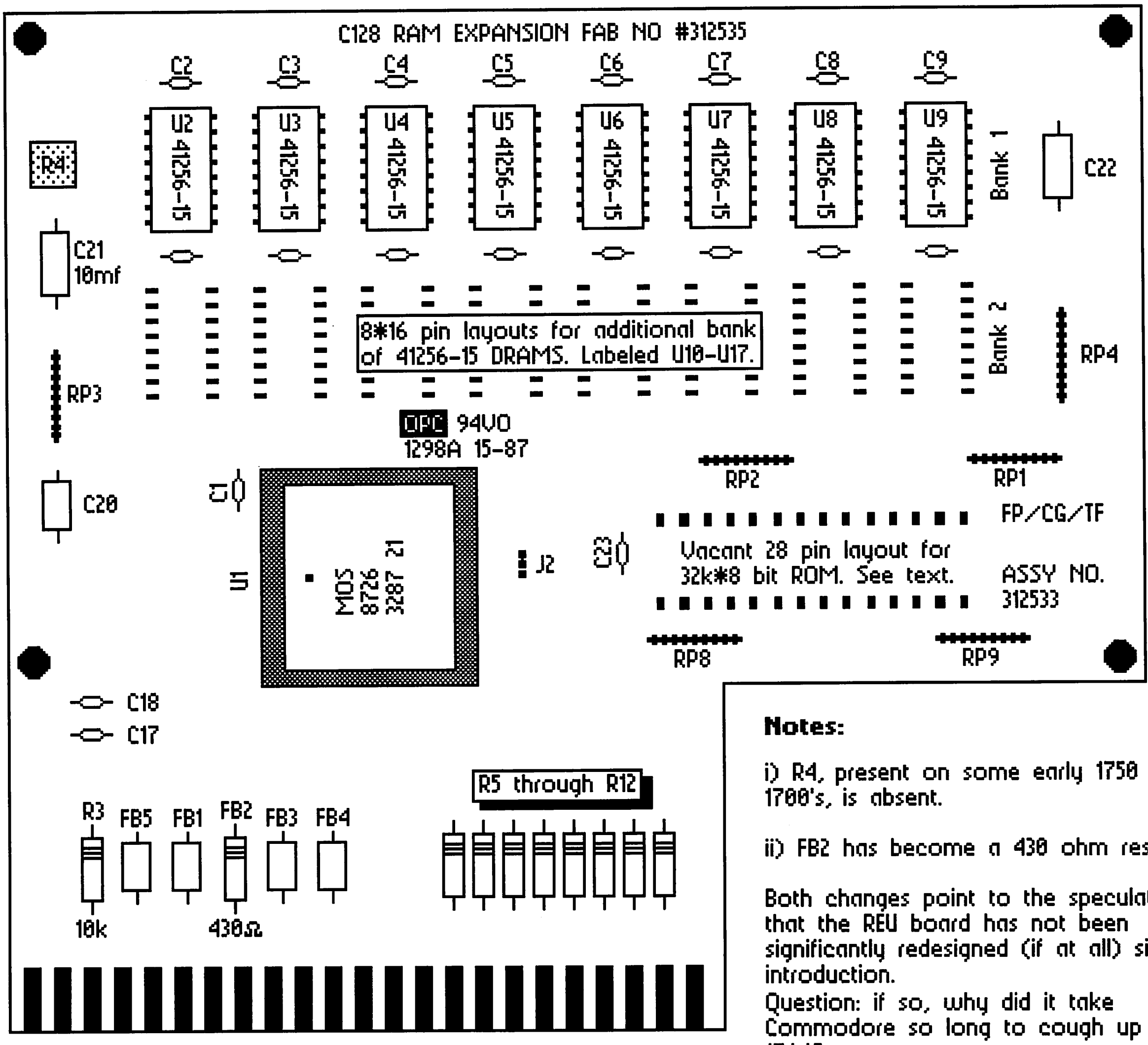
And that, ladies and gentlemen, is all there is to it. And, as a nice little bonus, your REU is (and always has been) compatible with the C128.

As for the EPROM, it's one of the more curious aspects of the REU. It seems that much was intended, but it was ultimately abandoned. All we're left with is the layout and the opportunity to ground /EXROM or both /GAME and /EXROM. At least, that would allow 8K or 16K of EPROM. But that seems pretty poor fare when 32K is an option, and that there might be lurking there, somewhere, a more elegant select mechanism.

Following up

Funny, the way things change. The above portion of this article was written in early January, just when it was becoming clear that a lot of people were interested in a one meg 64. It seemed obvious at the time that an article concerning the 1764 was again due. I knew no one had ever attacked the EPROM question, and I wanted to open a forum of sorts. Then in the February issue of *Commodore Magazine*, Brian Dougherty (of Berkeley Softworks) stated that an EPROM could just be

1764 Board Layout



Notes:

i) R4, present on some early 1750 and 1700's, is absent.

ii) FB2 has become a 430 ohm resistor.

Both changes point to the speculation that the REU board has not been significantly redesigned (if at all) since its introduction.
Question: if so, why did it take Commodore so long to cough up the 1764?

C64 EPROM Select Solutions

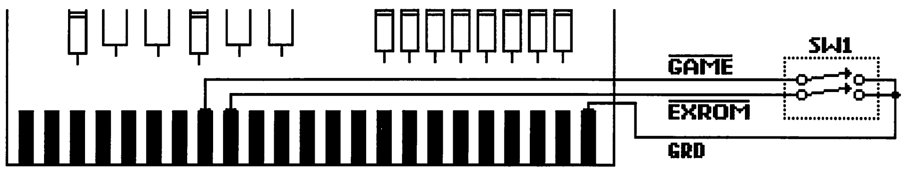
(for the 17xx REU's)

Notes:

i) REU is shown component side up.

ii) /GAME & /EXROM are leads 8 & 9 respectively.

iii) SW1 as shown is a DPST and when closed, maps a 16k EPROM into the $8000 to $bfff range. For an 8K EPROM use a SPST switch on the /EXROM line. The EPROM then maps in at $8000.



dropped into a 1764. Well, that got me going and the above is the result.

Now four months later, I still stand by the above conclusions. But, like I said, things have this funny habit of changing. And, we learn in the process. I missed a valuable point in the above notes, that something obvious I complained about: the 1764 is, down deep inside, still a product that was developed for the C128. And as such, the EPROM question needs to be re-examined. Unkown to me at that time was the select mechanism by which the C128 'logs in' external (and for that matter, internal) expansion ROM. For the uninitiated, and those who own 64's, I'll briefly sketch it out.

Ignoring the Z80 and its part in a 128's startup routine, we are left with only two routines in the 128's native mode: POLL and PHOENIX. Both routines are accessable through the much expanded KERNAL jump table. On startup or reset, POLL does just what its name suggests. First, the routine checks the state of the /GAME and /EXROM lines. If either is pulled low, a **go64** is executed and we end up in 64 mode. Failing that, the internal and external cartridge slots are polled for the ROM signature **CBM** (at ROMbase+7, where ROMbase equals either $8000 or $C000). If a cartridge is detected, its ID (ROMbase+6) is then logged in the Physical Address Table (PAT). If the ID is 1, then an auto-start cartridge is recognized, and its cold start entry (at ROMbase) is called immediately. Otherwise, this task is left to PHOENIX which checks the PAT and then calls the cold start routine of each cartridge logged there. That's pretty much it. The /GAME and /EXROM lines are considered only insofar as they indicate the presence of a C64 style cartridge. Interesting.

So what does all this mean? Basically this: the EPROM slot in the 17xx REU's was probably meant for the C128 only. Commodore's way of using the expansion port but leaving it free for further ROM expansion - clever! In a brief experiment, I dropped a 16K EPROM into my REU and plugged it into a C128. Result: mapped in at $88000 and $98000 (the first hex digit is the bank address) was my EPROM. None of the fancy select mechanisms suggested above, nothing. Just the 128's way of checking who's out there. That's why the /GAME and /EXROM lines are left unconnected. If they were, we'd end up with a **go64** and an REU that only worked on a C64 or in 64 mode. Maybe those unused bits do function just as I speculated four months ago. The format of a cartridge 'ROM signature' on the C128 and C64 differ significantly, making them incompatible. Perhaps some incompatibility issue is resolved through those bits. A cartridge designed for the 64 affects the 128 in ways not always desired, and a 128 cartridge wouldn't be recognised by the 64 at all! Those unused bits' function, though still hidden, may somehow be tied up in all this.And where does that leave us - the owners of these REUs? If you're a C64 owner then check out the following diagrams. A switch could easily be added to an REU that would allow an EPROM to be mapped in or out on power up or reset. If you're a C128 owner, just install a properly formatted EPROM.

Capitals: A Basic Quiz Program

Using linked lists

by Jim Butterfield

The program included at the end of this article runs on: Commodore 64, Commodore 128, Plus-4, Commodore 16, B128, PET, CBM.

Do you know the capital cities of the fifty states and ten provinces? This program will check your knowledge. It will give you full points for a correct answer; but if you miss with your first attempt, it will try to help you with multiple choices. And it gives hints.

Program *CAPITALS* is written in BASIC, and you might like to look at the code for some interesting programming methods. It uses carefully planned educational techniques; you might like to check these.

Since the program is in BASIC, it will be no speed demon. Considering the work it does, however, it clips along at an acceptable pace. If you happen to have access to a compiler, you may use this to speed up *CAPITALS*.

When you run the program, you'll find its operation to be fairly self-explanatory. The simple BASIC language doesn't protect against wild keyboard usage - for example, you could ramble around the screen using cursor movements when you were being asked for an input. But it does sensible things in most cases.

Educational considerations

The program presents states and provinces in 'shuffled' order. Each run will be different. As written, it will go through the entire list of 60 provinces and states, but the user may respond **end** at any time to terminate the quiz.

A hint is offered to help with a future question. The hint is set to give the answer to the question that is four items ahead. An attentive student's short-term memory will usually retain this. When this leads to a correct answer a little later, the information is reinforced; better learning takes place.

If the student misses the question, he or she is offered a multiple choice supplementary question. There are two kinds of multiple choice; which one the student sees will depend on the way the question was first answered.

If the student's reponse is wrong, but starts with the correct letter of the alphabet, the computer will present a list of all cities starting with that letter. Perhaps it was a spelling mistake; or, the student may have the right idea but the wrong answer. For example: suppose the computer asks for the capital of Kansas, and the student replies **Tulsa**. Since the correct answer starts with the letter T, the computer will list all cities starting with T.

On the other hand, if the student's response does not match - not even the first letter - the computer prepares a different kind of multiple choice. In this case, exactly six choices are presented. The computer has quite a search to find 'good' choices; there may be a short pause here. The choices will include the correct answer, of course, plus another city in the same state.

Whichever multiple choice method is used, the student is asked to type in the correct answer rather than a number or letter. It's good exercise, and will help the memory process.

If the student's first response to a question is wrong, but names a valid city known to the computer, the computer will say so. For example, if a student responds with **Tulsa** when asked for the capital of Kansas, the computer will advise that Tulsa is a city in Oklahoma. The student immediately receives the correct association for the city name.

The DATA List

You can shorten or lengthen the list of DATA items, up to a maximum of 99. You can replace it completely with another set of data, for example, European countries and their capitals. Data goes from line 100 to line 800; the last line says DATA END, which signals the computer that there are no more items.

The format of the data statements is easy to follow, but here are the details. The first item on each line is the state or province; then the capital city; then another city in that state. The second city is often chosen because it is large or well-known, but some, such as Springfield or Salem, might be picked because of a name that matches a capital city of a different state.

Program Details

This program carefully hooks together all cities whose name starts with the same letter of the alphabet. Thus, capital cities Phoenix, Providence and Pierre are linked, as are non-capital cities Pocatello, Philadelphia, and Portland. Using these 'linked lists', the program can rapidly search out multiple-choice candidates. More detail on linked lists is given below.

To vary the order of questions, a 'shuffle' must take place as the program starts. Moving strings around is a tedious business; instead, a table of 'quick pointers', array **Q()**, is shuffled. Later, array **Q** will tell us the order in which each state or province will be used. You may read the shuffling code in lines 1100 to 1140. Note that we use random function **rnd(0)** once only, to scramble the random sequence; after that, we use **rnd(1)** to generate unpredictable values.

The program starts at line 900, where it defines the arrays (tables and lists). Table **S$(state,column)** gives us the string values for each state: column 0 for the state name, column 1 for the capital city, and column 2 for the second city. Tables **A()** and **L()** are used to keep the linked lists; more about those in a moment. And list **Q()**, as we have mentioned, sets up the order in which we will ask the questions.

By line 1200, we've completed reading in the data and shuffling, and we can start asking questions. If the first answer is not correct, we'll call the subroutine at 3000 to do our multiple coice work for us. Line 1500 covers the ending summary.

The subroutine at 2000 offers the multiple choice menu to the student. The menu has been built in advance. This subroutine prints it, asks for the answer, and checks to see if the response is correct.

At 3000 is a subroutine that is used whenever the student has given a wrong initial answer. It decides which type of multiple choice question will be appropriate.

The subroutine at 4000 looks through lists of cities - both capitals and others. If the student's first response is not the correct answer, but is a valid city name, that information is printed. This subroutine also builds a multiple choice table of cities whose names start with the same letter as the input name. This table might be used, or it might be replaced by another multiple choice table; the decision will be made back in subroutine 3000.

Line 5000 contains a brief subroutine to add a city to the multiple choice list. Its main function is to remove duplicate city names.

At 6000 we have an elaborate subroutine to select multiple choice candidates. As an example: for state Arizona, where the capital is Phoenix and the other city is Tucson, the computer will pick three cities that start with P and three that start with T; three will be capitals and three not; and of course the list will include Phoenix and Tucson themselves.

Linked lists

This is a powerful programming method to hook similar items together. The program uses it to link cities whose names start with the same letter. That makes searches much faster: for example, to find all capital cities starting with the letter T, we don't need to search and compare all 50; instead, we follow the "T" chain.

Each city has, as part of its data, a pointer or 'link' to the next city that belongs in the group. At the end of the chain, there will be a zero pointer to say, "no more". To tell you where the chain starts, there are a set of starting pointers for each letter.

To find all the capital cities that start with letter T, for example, we look at the starting pointer for T (that's in array A, row 20 for letter T, column 1 for capital cities). That gives us the number of the first city in this chain. If it should happen that there are no cities starting with the selected letter (as is the case with the letter X, for example), we would get a value of zero.

To move on to the next city starting with that letter, we look at the pointer in array L (for "link"). It gives us the number of the next city, or a zero to signal "no more cities".

How do we build such linked lists? It's not hard. Before we read our data, we set all the starting pointers to zero. That, of course, means "no cities in this list" - so far.

When we read in a city, we pick out its first letter. We will go to the corresponding point in the starting table; in a moment, we'll put the identity of this new city there. But first, take the contents of that pointer and move it into the link of the new city. After that we make the entry in the starting table.

How does this work? If this is the first city beginning with a given letter, we'll pop its number into the starting table, and put the zero (from the starting table) into the city's link. Result: this city becomes the first in the linked list, and its link value of zero says that there are no more cities in the chain. Just what we want.

If, on the other hand, the new city is not the first that begins with that letter, our work with the starting table and link will add this city to the top of the chain. The starting table will now point at this new city, which in turn will point to the rest of the chain.

Linked lists are flexible, and may be used to hook together many type of data. In a genealogical data base, such a list might be used to build a chain of children in a given family; using this kind of data relationship means that there would be no fixed limit to the number of members of the family. In general data usage, a list of names linked by their first letter, similar to the system we have used here, can be a good way to search for a specific person; it would often be faster and more flexible than an alphabetized list search.

Conclusion

The program is a good way to test your skills. It may be easily modified to test other areas of knowledge. It uses valid educational methods to do more than quiz: it hints, it supplies extra information, and it gives the student more than one try for a correct answer.

And if you're interested in programming, you'll find a powerful technique here, linked lists, that can make your programs more flexible and efficient.

CAPITALS: *An educational program that demonstrates use of linked lists.*

```
AK 100 data california,sacramento,los angeles
JJ 110 data new york,albany,new york city
LP 120 data texas,austin,houston
NG 130 data pennsylvania,harrisburg,philadelphia
JB 140 data illinois,springfield,chicago
NO 150 data ohio,columbus,cleveland
ID 160 data florida,tallahassee,miami
AJ 170 data michigan,lansing,detroit
NN 180 data new jersey,trenton,jersey city
GK 190 data north carolina,raleigh,charlotte
FO 200 data massachusetts,boston,salem
PN 210 data georgia,atlanta,columbus
GJ 220 data virginia,richmond,norfolk
KM 230 data indiana,indianapolis,gary
DI 240 data missouri,jefferson city,saint louis
ED 250 data wisconsin,madison,milwaukee
ED 260 data tennessee,nashville,memphis
AJ 270 data louisiana,baton rouge,new orleans
GN 280 data maryland,annapolis,baltimore
CL 290 data washington,olympia,seattle
EF 300 data minnesota,saint paul,minneapolis
IE 310 data alabama,montgomery,birmingham
GL 320 data kentucky,frankfort,louisville
IP 330 data south carolina,columbia,charleston
NH 340 data oklahoma,oklahoma city,tulsa
BG 350 data connecticut,hartford,bridgeport
PK 360 data colorado,denver,colorado springs
EH 370 data iowa,des moines,cedar rapids
NE 380 data arizona,phoenix,tucson
JO 390 data oregon,salem,portland
NA 400 data mississippi,jackson,biloxi
EL 410 data kansas,topeka,kansas city
DP 420 data arkansas,little rock,fort smith
OD 430 data west virginia,charleston,huntington
PG 440 data nebraska,lincoln,omaha
GC 450 data utah,salt lake city,ogden
GI 460 data new mexico,santa fe,albuquerque
JK 470 data maine,augusta,portland
EC 480 data hawaii,honolulu,hilo
PL 490 data idaho,boise,pocatello
PF 500 data rhode island,providence,newport
DN 510 data new hampshire,concord,manchester
FA 520 data nevada,carson city,las vegas
HG 530 data montana,helena,billings
BJ 540 data south dakota,pierre,sioux falls
JA 550 data north dakota,bismark,fargo
FB 560 data delaware,dover,wilmington
MN 570 data vermont,montpelier,burlington
KN 580 data wyoming,cheyenne,casper
NF 590 data alaska,juneau,anchorage
PD 600 data ontario,toronto,ottawa
FH 610 data quebec,quebec city,montreal
MN 620 data british columbia,victoria,vancouver
PE 630 data alberta,edmonton,calgary
MC 640 data manitoba,winnipeg,brandon
PH 650 data saskatchewan,regina,saskatoon
KO 660 data nova scotia,halifax,sydney
CN 670 data new brunswick,fredericton,saint john
FL 680 data newfoundland,saint john's,gander
BP 690 data prince edward island,charlottetown,summerside
BH 800 data end
GC 900 dim s$(99,2),l(99,2),q(99),a(26,2)
AB 910 dim c(20)
NI 920 j=0:print"please wait for data load"
AI 930 j=j+1
PJ 940 read s$(j,0):if s$(j,0)="end" goto 1040
MI 950 read s$(j,1),s$(j,2)
JH 960 for k=0 to 2
MG 970 a=asc(s$(j,k))-64
KG 980 if a<1 or a>26 then print s$(j,k);"?":stop
ML 990 l(j,k)=a(a,k)
FD 1000 a(a,k)=j
HN 1010 next k
JK 1020 q(j)=j
PL 1030 goto 930
MF 1040 s9=j-1
FO 1100 j=rnd(0)
EF 1110 for j=1 to s9
IC 1120 k=int(rnd(1)*s9)+1
HM 1130 q=q(k):q(k)=q(j):q(j)=q
GF 1140 next j
PJ 1150 print chr$(147);chr$(142)
OK 1160 print "capital city quiz"
BM 1170 print "                jim butterfield"
OK 1200 for j=1 to s9
BP 1210 q1=j+4
HD 1220 if q1<s9 then print "hint:  capital of ";s$(q(q1),0);" is ";s$(q(q1),1)
AA 1230 print
FP 1240 q0=q(j)
NB 1250 print "what is the capital of ";s$(q0,0);"?"
ID 1260 r$="#":input r$:print chr$(142);:if r$="end" goto 1500
FM 1270 m=0:m1=0
AL 1280 if r$=s$(q0,1) then m=10
IA 1290 r=asc(r$):r0=r-64
KO 1300 if m=0 then gosub 3000
BJ 1310 if m=0 then print  "no points!  answer is ";s$(q0,1)
LB 1320 s=s+m:if m>0 then print "right!";:if m<10 then print " (for part points)";
GH 1330 print:print
PN 1340 s0=s0+10
IA 1350 print "score: ";s;" out of possible ";s0
GF 1360 if j/10=int(j/10) then print "(reply 'end' to quit)"
MD 1370 next j
OG 1380 goto 1510
GB 1500 print "(answer: ";s$(q0,1);")"
GI 1510 print "your score:"
AM 1520 s$=" a very poor":if s0=0 goto 1590
GF 1530 if s/s0>.5 then s$=" a mediocre"
BM 1540 if s/s0>.7 then s$="* a passable"
DI 1550 if s/s0>.8 then s$="** a decent"
PH 1560 if s/s0>.9 then s$="*** a good"
NG 1570 if s/s0>.95 then s$="**** a fantastic"
ED 1580 if s=s0 then s$="***** a perfect"
DP 1590 print s$;s;"out of";s0
AE 1600 end
EG 1999 rem: offer choice menu
MC 2000 m1=1:print:print"the capital of ";s$(q0,0);" is:"
KI 2010 for j1=1 to c
ID 2020 k=int(rnd(1)*c)+1
HD 2030 q=c(j1):c(j1)=c(k):c(k)=q
OJ 2040 next j1
CL 2050 for j1=1 to c
NF 2060 c0=c(j1):c1=1
BC 2070 if c0<0 then c0=0-c0:c1=2
DO 2080 print "     ";s$(c0,c1)
AN 2090 next j1
```

```
GG  2100 print
NI  2110 print "type in the answer, correctly spelled.."
MK  2120 x$="#":input x$:print chr$(142);:if x$="end" goto 1500
AD  2130 if x$=s$(q0,1) then m=m0
IH  2140 return
EE  2999 rem:  answer search strategy
AD  3000 r1=asc(s$(q0,1))-64:r2=asc(s$(q0,2))-64
CH  3010 rem: search name for any match
PK  3020 if r0>0 and r0<26 then gosub 4000
IG  3030 rem: first letter matches; menu
HE  3040 if m=0 and r0=r1 then m0=7:gosub 2000
GK  3050 rem: no luck, try general menu
DB  3060 if m=0 and m1=0 then gosub 6000
KB  3070 return
NO  3999 rem: check for any match, build table
MP  4000 c=0
AG  4010 l=a(r0,1):l0=1:l1=1
LK  4020 if l=0 goto 4070
HB  4030 if r$=s$(l,1) then print r$;" is the capital of ";s$(l,0);"!"
BN  4040 gosub 5000
BO  4050 l=l(l,1)
MN  4060 goto 4020
OL  4070 l=a(r0,2):l0=-1:l1=2
PP  4080 if r$=s$(l,2) then print "..";r$;" is a city in ";s$(l,0);"!"
GA  4090 if l=0 or c>15 goto 4130
NA  4100 gosub 5000
BC  4110 l=l(l,2)
OB  4120 goto 4080
OD  4130 return
IM  4999 rem: build table of non dup names
FG  5000 if c=0 goto 5050
CE  5010 for j1=1 to c
LD  5020 l2=1:if c(j1)<0 then l2=2
HC  5030 if s$(abs(c(j1)),l2)=s$(l,l1) goto 5070
GF  5040 next j1
OG  5050 c=c+1
LJ  5060 c(c)=l*l0
KO  5070 return
LL  5999 rem: build general multiple choice
CB  6000 c=2:c(1)=q0:c(2)=-q0
DD  6010 l=a(r1,1):l0=1:l1=1
AI  6020 if l=0 goto 6060
HJ  6030 gosub 5000
HK  6040 l=l(l,1)
IK  6050 goto 6020
CI  6060 if c=2 then l=int(rnd(1)*s9)+1:gosub 5000:goto6060
NN  6070 c1=c(int((c-2)*rnd(1))+3)
GE  6080 c=3:c(3)=c1
FK  6090 l=a(r1,2):l0=-1:l1=2
KO  6100 if l=0 or c>15 goto 6140
HO  6110 gosub 5000
LP  6120 l=l(l,2)
KP  6130 goto 6100
PM  6140 if c=3 then l=int(rnd(1)*s9)+1:gosub 5000:goto 6140
CD  6150 c1=c(int((c-3)*rnd(1))+4)
OJ  6160 c=4:c(4)=c1
GN  6170 l=a(r2,1):l0=1:l1=1
EC  6180 if l=0 goto 6220
HD  6190 gosub 5000
HE  6200 l=l(l,1)
CF  6210 goto 6180
MB  6220 if c=4 then l=int(rnd(1)*s9)+1:gosub 5000:goto 6220
HI  6230 c1=c(int((c-4)*rnd(1))+5)
GP  6240 c=5:c(5)=c1
IE  6250 l=a(r2,2):l0=-1:l1=2
EI  6260 if l=0 or c>15 goto 6300
HI  6270 gosub 5000
LJ  6280 l=l(l,2)
EK  6290 goto 6260
JG  6300 if c=5 then l=int(rnd(1)*s9)+1:gosub 5000:goto 6300
MN  6310 c1=c(int((c-5)*rnd(1))+6)
OE  6320 c=6:c(6)=c1
EO  6330 m0=5:gosub 2000
AO  6340 return
```

C Problems, Tips and Observations

Compiler anomalies and drive usage

by Larry Gaynier

I have the *C Power* compilers for both the C64 and the C128. Since I first purchased the *C Power* compiler, I have discovered some problems that I will share with you. Overall, I am favourably impressed by the package. Hopefully, I can help you avoid future aggravation and wasted effort if you come up against one of these problems. In this article, I assume *Power C* from Spinnaker is identical to *C Power* from Pro-Line.

Command line arguments (C128 and C64)

Keep these limits in mind when you use command line arguments, especially on the C128. The C64 shell supports 22 arguments; argv[0] through argv[21]. I have found this to be adequate. The C128 shell is more restrictive, supporting only 10 arguments; argv[0] through argv[9]. I find this to be inadequate and aggravating when using programs that allow lots of switches and arguments. Exceeding the limit can lock up the machine, which may not recover after a soft reset (RUN/STOP-RESTORE).

A compiler bug (C128 and C64)

The C compiler has an elusive bug that was very difficult to track down. Originally, I noticed inconsistent results in a program that used the following IF statement:

```
if ((ptr->c = calloc (10,1)) != 0)
```

In this example, **ptr** points to a structure containing **c**, a pointer to char. The result of the call to CALLOC is assigned to the structure variable **c**. The IF statement tests the assignment result to see if a non-zero pointer address was returned by CALLOC. CALLOC returns zero if there is not enough free memory to satisfy the request. In situations where plenty of free memory was available, my program would randomly behave as if CALLOC had returned zero. After months of haphazard research into the problem, I discovered it had something to do with the indirection operator * (**ptr->c** is shorthand for (***ptr).c**).

The *bug.c* program (Listing 1 at the end of this article) demonstrates the basic problem. The key statement in this program is the multiple assignment. The integer **i2** and the integer pointed to by **ip** are to be set to the integer **i1**. One would expect **i1**, ***ip** and **i2** to always be equal. This is based on the consequence that the result of any assignment in C has a value that is available for subsequent use. Here is a sample of the output from this program:

```
i1 = -512   *ip = -512   i2 = 0
i1 = -511   *ip = -511   i2 = 1
i1 = -510   *ip = -510   i2 = 2
i1 = -509   *ip = -509   i2 = 3
   .
   .
i1 = -258   *ip = -258   i2 = 254
i1 = -257   *ip = -257   i2 = 255
i1 = -256   *ip = -256   i2 =   0
i1 = -255   *ip = -255   i2 =   1
i1 = -254   *ip = -254   i2 =   2
   .
   .
i1 = -3     *ip = -3     i2 = 253
i1 = -2     *ip = -2     i2 = 254
i1 = -1     *ip = -1     i2 = 255
i1 =  0     *ip =  0     i2 =   0
i1 =  1     *ip =  1     i2 =   1
i1 =  2     *ip =  2     i2 =   2
i1 =  3     *ip =  3     i2 =   3
   .
   .
i1 = 253    *ip = 253    i2 = 253
i1 = 254    *ip = 254    i2 = 254
i1 = 255    *ip = 255    i2 = 255
i1 = 256    *ip = 256    i2 =   0
i1 = 257    *ip = 257    i2 =   1
i1 = 258    *ip = 258    i2 =   2
i1 = 259    *ip = 259    i2 =   3
   .
   .
i1 = 510    *ip = 510    i2 = 254
i1 = 511    *ip = 511    i2 = 255
i1 = 512    *ip = 512    i2 =   0
```

This clearly shows that something is wrong. The only values that seem to be correct are in the range 0 to 255. All negative values and values greater 255 produce bad results. The cause of this bug can be seen when sample code is disassembled.

```
main()
{
   int *ip, i1, i2;

   i2 = *ip = i1;
}
```

This simple example produces the following C128 object code after compilation:

```
main
85 fb 00 / 1803 sta $fb
a9 06 00 / 1805 lda #$06
a2 00 00 / 1807 ldx #$00
a0 00 00 / 1809 ldy #$00
20 20 20 / 180b jsr c$105      function preparation
a6 04 00 / 180e ldx $04        get low byte of i1
a4 05 00 / 1810 ldy $05        get high byte of i1
    begin indirect *ip
98 00 00 / 1812 tya            $02,$03 has address
a0 01 00 / 1813 ldy #$01
91 02 00 / 1815 sta ($02),Y    save high byte indirect
8a 00 00 / 1817 txa
88 00 00 / 1818 dey            at this point Y is zero
91 02 00 / 1819 sta ($02),Y    save low byte indirect
    end indirect *ip
    Y incorrectly assumed
    to contain the high byte
86 06 00 / 181b stx $06        save low byte of i2
84 07 00 / 181d sty $07        save high byte of i2
    always zero
a9 06 00 / 181f lda #$06
a2 00 00 / 1821 ldx #$00
a0 00 00 / 1823 ldy #$00
4c 4c 4c / 1825 jmp c$106      function wrap-up
```

The C64 object code is identical except for zero page locations. For integer-type variables, the compiler assumes the result of any assignment remains available in the X,Y registers (low byte, high byte). During the indirect assignment through a pointer, the Y register is used for indirect addressing. But nothing is done to restore the high byte to the Y register. As a result, the high byte is always zero.

The compiler should restore the high byte after saving the low byte. A sample fix is:

```
iny
lda ($02),Y
tay
```

It should be inserted after the second **sta ($02),Y** instruction. This fix would make the compiler behave like 'standard' C at the expense of adding four bytes to all indirect assignments.

I advise you not to use the result of an indirect assignment. Typically, the offending line must be broken up into separate statements. The restriction only applies to signed and unsigned integer variables. Character variables do not exhibit the problem because the upper byte is naturally zero. Float variables are handled differently through special subroutines. My earlier examples can be rewritten to avoid the problem:

```
i2 = *ip = i1;
```

can be rewritten as:

```
*ip = i1;
i2 = *ip;
```

and,

```
if ((ptr->c = calloc (10,1)) != 0)
```

can be rewritten as:

```
ptr->c = calloc (10,1);
if (ptr->c != 0)
```

PEEK, POKE, SYS (C128 only)

At the heart of the C128 is a powerful bank switching scheme to select between RAM banks, BASIC ROM, Kernal ROM, and the I/O registers. Additional capabilities include zero page relocation and sharing common RAM at the top or bottom of the RAM banks. These features are controlled by the memory management unit (MMU) with configuration registers located at $D500-$D50B and $FF00-$FF04. These registers are critical to the correct operation of the C128 and require careful manipulation. Otherwise, a program may lose control of the machine and appear to be locked up.

The C environment makes full use of the C128 banking features. The shell and RAM disk reside in RAM bank 0, while programs reside and execute from RAM bank 1. The lowest 1K of memory is shared between both banks. During program execution, the memory page (256 bytes) at $1300 is used as zero page RAM. The first 32 bytes of automatic variables declared in a function are placed in the zero page. The true zero page is switched back as zero page RAM for any calls to routines that do not reside within the program, such as the Kernal, BASIC or shell routines. The true zero page contains many registers needed for calling the Kernal or BASIC routines.

The PEEK, POKE, and SYS functions supplied with the function library have been appropriately modified for C128 memory management. However, there is a problem that can be easily demonstrated by the PEEKTEST.C program (Listing 2). This program peeks a byte from any C128 bank. When run, it produces strange results. You can easily see this if you attempt to peek locations with known values such as the Kernal jump table. The problem lies in the PEEK and POKE functions.

But first, some background. The shell and C programs use two special routines in common RAM to switch control between C128 banks.

The subroutine at $0124 does a JSRFAR to any bank.

```
ad 00 ff / 0124 lda $ff00
48 00 00 / 0127 pha        save configuration
a9 00 00 / 0128 lda #$00
8d 00 ff / 012a sta $ff00  set bank 15
20 6e ff / 012d jsr $ff6e  JSRFAR
68 00 00 / 0130 pla
8d 00 ff / 0131 sta $ff00  restore configuration
60 00 00 / 0134 rts
```

The zero page registers $02-$08 must be set up according to JSRFAR preparation before the call to $0124.

The subroutine at $0135 stores a value to a configuration register.

```
a8 00 00 / 0135 tay
ad 00 ff / 0136 lda $ff00
48 00 00 / 0139 pha          save configuration
a9 00 00 / 013a lda #$00
8d 00 ff / 013c sta $ff00    set bank 15
98 00 00 / 013f tya
9d 00 d5 / 0140 sta $d500,X  set config register
68 00 00 / 0143 pla
8d 00 ff / 0144 sta $ff00    restore configuration
60 00 00 / 0147 rts

a = configuration value to store
x = offset from $D500, which determines the
    configuration register to be updated.
y = used internally by the $0135 routine.
```

The basic procedure to execute a routine in another bank is:

1) Switch zero page back to $0000 using the $0135 routine.

2) Set up registers $02-$08 for the Kernal routine JSRFAR.

3) Call $0124 in common RAM which calls JSRFAR.

4) Capture results from registers $02-$08 as appropriate.

5) Switch the zero page to $1300 using the $0135 routine.

The problem in the PEEK and POKE functions is that they fail to set up the X register before the call to $0135. The net result is that zero page does not get properly swapped. If a function invokes PEEK or POKE, the first six bytes of variables declared within the function will be corrupted because the zero page locations $02-$08 are overwritten during the setup and execution of JSRFAR. The program POKE.A (Listing 3) contains the updated source code for the PEEK and POKE functions. Old and new code is highlighted.

The SYS function does not exhibit the zero page swapping problem like PEEK and POKE. However, SYS does directly manipulate the configuration registers. One problem of direct manipulation of the configuration registers is that an executing program may be switched out and lose control. Since I prefer robust and consistent designs, I modified SYS to eliminate all direct manipulation of the configuration register, making it behave like PEEK and POKE. The program SYS.A (Listing 4) contains the updated source code for the SYS function. Old and new code is indicated.

PEEK.A and SYS.A can be assembled using any assembler that produces compatible C object. Alternatively, BASIC generator programs (Listings 5 and 6) have been supplied to create the object files for PEEK, POKE and SYS using DATA statements. Delete the original object files PEEK.OBJ and SYS.OBJ on your function library disk and replace them with the updated object files. Make sure the new object files use the same file names as the originals. Otherwise, you will need to use a object library editor to change the file names in the object libraries. As a precaution, use a backup copy for this and save the original function library disk.

A painful experience (C128 only)

I never paid much attention to magazine descriptions of 1571 disk drive problems. The information always seemed too vague to apply to me. I assume the problem that I painfully discovered came as a result of one of the 1571 bugs. It strongly reinforced my habit of doing periodic back-ups and organizing disks to minimize my reliance on any one disk.

When I first obtained the *C Power* compiler for my C128, I configured my C environment using two 1571 disk drives as follows:

1) Work drive 0 was set to disk unit 9 and held the floppy disk containing programs under development.

2) System drive 1 was set to the RAM disk unit 7. As part of my startup procedure, I copied the compiler, editor and various utilities to the RAM disk, as much as would fit. Then I would switch the system drive 1 between the RAM disk and disk unit 8 depending upon what utility I needed.

With this approach, editing and compiling were done to and from disk. I adopted this configuration because it was the fastest way to load the compiler, about two seconds. C128 compile time basically amounted to reading the source program from disk and writing back the object. These delays could be substantial when you consider the writing speed of 1571 is identical to the 1541. But I had been conditioned by the slow loading time of the C64 version of the compiler operating from a 1541 disk drive. I operated comfortably in this mode for about six months before I encountered the problem.

The problem occurs when the disk block total crosses the side 1 to side 2 boundary. The compiler can go into an infinite loop, eating up disk space, and corrupting the source file and possibly other files. When the problem occurred, I had to reset the computer, leaving the disk corrupt. After validating the disk, I found two 'splat' files TEMPXXX1 and TEMPXXX2. I assume the TEMPXXX files were written by the compiler. In addition, the source file was completely corrupted.

I saw similar problems when a C program was executing under the following conditions:

1) The disk block total was at or near to the side 1 to side 2 boundary.

2) The error channel was open.

3) Two files were opened by *fopen*.

4) Command line I/O redirection sent the standard output to a disk file.

Notice the total is three open files plus the error channel on one disk drive, which is the limit according to the *1571 User's Guide*. Although I do not have the details, I recall that one of the 1571 bugs is related to the side 1 to side 2 transition.

[Bugs corrected by the new 1571 ROM are documented in the ***C128 Developer's Package*** *from Commodore. - Ed.]*

After further investigation, I concluded that the problem never occurs with anything less than three open files. Any program using two open files or less plus the error channel appears to operate correctly across the side 1 to side 2 boundary. The C compiler may have produced the same situation with the error channel plus three open files: TEMPXXX1, TEMPXXX2, and either the source file or the object file.

Working around the bug

I considered many alternatives to avoid this problem. One solution was to keep disk space low to prevent crossing the side 1 to side 2 boundary. However, this defeated the advantage of a double-sided disk drive.

The approach I eventually took was to avoid all disk drive situations that would have three open files plus the error channel. I did this by making better use of the RAM disk.

After some experimenting, I finally configured my C environment as follows:

1) Work drive 0 is set to the RAM disk unit 7 and holds the programs under development.

2) System drive 1 is set to the disk unit 8 and holds the floppy disk containing the compiler, editor and various utilities.

3) Drive 2 is set to the disk unit 9 and holds the floppy disk containing programs under development.

With this approach, editing and compiling are done to and from the RAM disk. The procedure is simple and quick. Keep in mind that the normal search order for programs and files is work drive 0 followed by system drive 1, unless the drive number is explicitly given in the file name. The basic procedure is:

1) Set up the RAM disk configuration using the commands **rdon**, **setu 0 7 0** and **setu 2 9 0**.

2) Copy the appropriate files from drive 2 (floppy) to drive 0 (RAM) using the command **cp 2:file 0**. Be sure to copy any header files needed by the source files.

3) Edit the files using the command **ed file**. You can even pull files directly into the editor from drive 2 (floppy) using the command **ed 2:file** or using the command **get 2:file** from within the editor. The editor command **put file** writes the file back to drive 0 (RAM). Similarly, the command **put 2:file** writes the file directly back to drive 2 (floppy).

4) Compile the file from drive 0 (RAM) using the command **cc file**. The object file will be written to drive 0 (RAM).

5) When finished, delete the old copies from drive 2 using the command **rm 2:file**. Then, copy the updated files from drive 0 (RAM) back to drive 2 (floppy) using the command **cp 0:file 2**. Wildcard characters in the file name are very useful here. Be careful! It is very easy to lose hours of work during this step.

This configuration is quite fast. I/O to the RAM disk is done at blinding speeds by 1541 standards. Any delays come from loading the editor, compiler or other utilities. However, I consider these delays to be minor since the programs are loaded in 1571 burst mode. For example, the compiler takes about 10 seconds to load and the translator for the compiler takes about 5 seconds.

Important: Don't forget to copy files back to drive 2 (floppy) when your work is complete. All changes made to files residing in drive 0 (RAM) will be lost when the power is turned off.

Listing 1: *bug.c*

```
/*
** bug.c
**
** demonstrate C compiler bug
**
** Larry J. Gaynier
** June 25, 1988
*/

main()
{
  int  *ip, i0, i1, i2;

  ip = &i0;

  for (i1 = -512; i1 <= 512; i1++)
    {
      i2 = *ip = i1;
      printf ("i1 = %d    *ip = %d    i2 = %d\n", i1, *ip, i2);
    }
}
```

Listing 2: *peektest.c*

```
/*
** peek an address from a C128 bank
**
** Larry J. Gaynier
** June 25, 1988
*/

main (argc, argv)
unsigned argc;
char **argv;
{
   unsigned bank, address;
   char byte, peek();

   if (argc == 3)
   {
     sscanf (*++argv, "%x", &bank);
     sscanf (*++argv, "%x", &address);

     byte = peek (bank, address);
     printf ("bank = %02x    address = %04x    byte = %02x\n"
             ,bank, address, byte);
   }
   else
   {
     prusage();
     exit();
   }
}

prusage()
{
  printf ("usage: peektest bank address\n");
}
```

Listing 3: *peek.a* - revised source for *peek.o*

```
;modified to correct zero paging
;added ldx #$07 before every jsr $0135
;
;Larry Gaynier
;March 10,1987
;--------------------
 .def peek, poke
 .ref c$funct_init
;--------------------
peek
 jsr c$funct_init
 stx $a0
 txa
 pha
 lda #$00
;---- new ----------- old ----
 ldx #$07
;---- end ----------- end ----
 jsr $0135
 pla
 tax
 lda $0402,X
 sta $fc
 lda $0403,X
 sta $fd
 lda #$0f
 sta $02
 lda #$ff
 sta $03
 lda #$74
 sta $04
 php
 pla
 sta $05
 lda #$fc
 sta $06
 lda $0400,X
 sta $07
 lda #$00
 sta $08
 jsr $0124
 lda $06
 pha
 lda #$13
;---- new ----------- old ----
 ldx #$07
;---- end ----------- end ----
 jsr $0135
 pla
 ldx $a0
 sta $0400,X
 lda #$00
 sta $0401,X
 rts

;--------------------
poke
 jsr c$funct_init
 stx $a0
 txa
 pha
 lda #$00
;---- new ----------- old ----
 ldx #$07
;---- end ----------- end ----
 jsr $0135
 pla
 tax
 lda $0402,X
 sta $fc
 lda $0403,X
 sta $fd
 lda #$fc
 sta $02b9
 lda #$0f
 sta $02
 lda #$ff
 sta $03
 lda #$77
 sta $04
 php
 pla
 sta $05
 lda $0404,X
 sta $06
 lda $0400,X
 sta $07
 lda #$00
 sta $08
 jsr $0124
 lda #$13
;---- new ----------- old ----
 ldx #$07
;---- end ----------- end ----
 jmp $0135
```

Listing 4: *sys.a* - revised source for *sys.o*

```
;modified to use $0135 routine like peek and poke
;eliminated resetting the configuration register
;no need to enable the kernal and i/o
;handled by the $0135 routine
;
;Larry Gaynier
;March 14, 1987
;--------------------
 .def sys
 .ref c$funct_init
;--------------------
```

```
sys
 jsr c$funct_init
 stx $a0
 lda $0404,X
 sta $a2
 lda $0405,X
 sta $a3
 lda $0406,X
 sta $a4
 lda $0407,X
 sta $a5
 lda $0408,X
 sta $a6
 lda $0409,X
 sta $a7
 ldy #$00
 lda ($a2),Y
 pha
 lda ($a4),Y
 pha
 lda ($a6),Y
 pha
;-- new ------------- old ----
 txa                 lda #$4e
 pha                 sta $ff00
 lda #$00            lda #$00
 ldx #$07            sta $d507
 jsr $0135
 pla
 tax
;-- end ------------- end ----
 lda $0400,X
 sta $02
 lda $0403,X
 sta $03
 lda $0402,X
 sta $04
 pla
 sta $08
 pla
 sta $07
 pla
 sta $06
 php
 pla
 sta $05
 jsr $0124
 lda $05
 pha
 lda $06
 pha
 lda $07
 pha
 lda $08
 pha
 lda #$13
;-- new ------------- old ----
 ldx #$07            sta $D507
 jsr $0135           lda #$7F
;                    sta $ff00
;-- end ------------- end ----
 ldy #$00
 pla
 sta ($a6),Y
 pla
 sta ($a4),Y
 pla
 sta ($a2),Y
 plp
 lda #$00
 bcc skip
 lda #$01
skip
 ldx $a0
 sta $0400,X
 lda #$00
 sta $0401,X
 rts
```

Listing 5: *peek.gen* - a BASIC generator for *peek.o*

```
BN  100 rem  generator for "peek.o"
JN  110 n$="peek.o": rem  name of program
DC  120 nd=209: sa=151: ch=15837
```

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

```
FD  1000 data  32,   0,   0, 134, 160, 138,  72, 169
BP  1010 data   0, 162,   7,  32,  53,   1, 104, 170
HD  1020 data 189,   2,   4, 133, 252, 189,   3,   4
MK  1030 data 133, 253, 169,  15, 133,   2, 169, 255
HB  1040 data 133,   3, 169, 116, 133,   4,   8, 104
FD  1050 data 133,   5, 169, 252, 133,   6, 189,   0
CP  1060 data   4, 133,   7, 169,   0, 133,   8,  32
EN  1070 data  36,   1, 165,   6,  72, 169,  19, 162
JD  1080 data   7,  32,  53,   1, 104, 166, 160, 157
KL  1090 data   0,   4, 169,   0, 157,   1,   4,  96
JJ  1100 data  32,   0,   0, 134, 160, 138,  72, 169
FF  1110 data   0, 162,   7,  32,  53,   1, 104, 170
LJ  1120 data 189,   2,   4, 133, 252, 189,   3,   4
JL  1130 data 133, 253, 169, 252, 141, 185,   2, 169
EI  1140 data  15, 133,   2, 169, 255, 133,   3, 169
GJ  1150 data 119, 133,   4,   8, 104, 133,   5, 189
CK  1160 data   4,   4, 133,   6, 189,   0,   4, 133
HL  1170 data   7, 169,   0, 133,   8,  32,  36,   1
CN  1180 data 169,  19, 162,   7,  76,  53,   1,   0
KN  1190 data   0,   2,   0,  80,  69,  69,  75,   0
GO  1200 data   1,   0,   0,  80,  79,  75,  69,   0
PM  1210 data   1,  80,   0,   2,   0,  67,  36,  70
DM  1220 data  85,  78,  67,  84, 164,  73,  78,  73
EN  1230 data  84,   0,   0,   0,   0,   0,  67,  36
CM  1240 data  70,  85,  78,  67,  84, 164,  73,  78
JM  1250 data  73,  84,   0,   0,   0,  80,   0,   0
CI  1260 data   0
```

Listing 6: *sys.gen* - a BASIC generator for *sys.o*

```
JM  100 rem  generator for "sys.o"
GH  110 n$="sys.o": rem  name of program
PA  120 nd=168: sa=136: ch=13130
```

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

```
IA  1000 data  32,   0,   0, 134, 160, 189,   4,   4
CE  1010 data 133, 162, 189,   5,   4, 133, 163, 189
ID  1020 data   6,   4, 133, 164, 189,   7,   4, 133
OH  1030 data 165, 189,   8,   4, 133, 166, 189,   9
JP  1040 data   4, 133, 167, 160,   0, 177, 162,  72
DA  1050 data 177, 164,  72, 177, 166,  72, 138,  72
IC  1060 data 169,   0, 162,   7,  32,  53,   1, 104
KF  1070 data 170, 189,   0,   4, 133,   2, 189,   3
BF  1080 data   4, 133,   3, 189,   2,   4, 133,   4
GC  1090 data 104, 133,   8, 104, 133,   7, 104, 133
EF  1100 data   6,   8, 104, 133,   5,  32,  36,   1
FI  1110 data 165,   5,  72, 165,   6,  72, 165,   7
CG  1120 data  72, 165,   8,  72, 169,  19, 162,   7
AH  1130 data  32,  53,   1, 160,   0, 104, 145, 166
EH  1140 data 104, 145, 164, 104, 145, 162,  40, 169
OH  1150 data   0, 144,   2, 169,   1, 166, 160, 157
AA  1160 data   0,   4, 169,   0, 157,   1,   4,  96
IH  1170 data   0,   0,   1,   0,  83,  89,  83,   0
NF  1180 data   1,   0,   0,   1,   0,  67,  36,  70
FK  1190 data  85,  78,  67,  84, 164,  73,  78,  73
KC  1200 data  84,   0,   0,   0,   0,   0,   0,   0
```

Programming GEOS Icons

Some tricks for using more than 31 icons

by James Cook

One of the most fundamental user interface tools in the GEOS operating system is the icon. An icon is a graphic image on the monitor that causes a program routine to be called when the user clicks the mouse pointer over it. Icons, along with the mouse and pulldown menus, allow the user to operate the computer in a comfortable, easy to grasp way. Loading and running a program is as simple as moving the pointer over the file icon on the DeskTop and double-clicking. The user does not need to remember and type complicated, often confusing commands.

GEOS-specific application programs make extensive use of the GEOS Kernal, which is loaded from the boot disk and consists of a large number of service routines. These service routines take care of most of the common needs of an application such as disk and file handling, graphic manipulation, the user interface, etc. The memory-resident routines wait patiently for a user event to occur, such as clicking on an icon, before swinging into action. The GEOS programmer can freely use these routines in any GEOS application.

Suprisingly, as important as icons are to GEOS, there is only one icon-specific routine, *DoIcons*. This single routine, however, packs a lot of punch. All of the file and disk icons on the DeskTop use it. All of the tool routines in *geoPaint* are called by it. So important is this routine that Berkeley Softworks' *Official GEOS Programmers Reference Guide* warns that GEOS assumes an application will always have at least one icon. Since most applications depend heavily on the use of icons, it is worth an in-depth look at how to use them in your own programs.

The *DoIcons* routine is almost always a part of an application's initialization sequence. This sequence calls a number of GEOS graphics and menu routines to create the user interface screen. These routines, as well as most of the other available service routines, are accessed by a JSR to an entry in a jump table. The jump table contains the actual address in memory of the routine the programmer wishes to access. While different versions of GEOS may locate the service routines at different addresses, the jump table remains the same between versions. Since GEOS has been frequently updated, the jump table provides a stable path to the routine. The GEOS jump table address for *DoIcons* is $C15A.

When *DoIcons* is called, the GEOS Kernal expects the two-byte **.word** following the JSR in memory to contain the pointer to the icon table. The icon table tells GEOS how many icons are required, their location on the screen, their size, the location of the compacted bit-map image for each icon, the service routine to call after the icon is selected, and the position to place the mouse pointer after the icons are drawn. Here is an example of *DoIcons* and the icon table:

```
;Icon table example
JSR    DoIcons    ;Call the icon routine.
.word IconTable ;Pointer to the icon table.
IconTable:        ;Here is the icon table.

.byte   8         ;8 icons on the screen.
.word   160       ;The mouse to be in the center of
.byte   100       ;screen after icons are drawn.
.word   IconPic1  ;Pointer to bit-map for icon #1.
.byte   10        ;X Position of the icon in bytes.
.byte   10        ;Y Position of the icon in pixels.
.byte   2         ;Width of icon in bytes.
.byte   10        ;Height of icon in pixels.
.word   DoIcon1   ;Pointer to service routine for #1.
.word   IconPic2  ;Pointer to bit-map for icon #2.
.byte   12        ;X Position of the second icon.
.byte   10        ;Y Position of the second icon.
.byte   2         ;Width of icon in bytes.
.byte   10        ;Height of icon in pixels.
.word   DoIcon2   ;Pointer to service routine for #2.
. . . .   . . .   ;6 more icon entries.
```

Let's take a closer look at the icon table. As noted, every application must have at least one icon so there will always be at least one call to *DoIcons* and an icon table in every application. The minimum number in the first entry of the icon table, therefore, is 1. The maximum number of active icons possible on the screen is 31.

You may call *DoIcons* several times in an application, but only the most recently called group of icons will be active. GEOS will not erase the previously drawn icon graphics from the screen. You may re-call a *DoIcons* and reactivate its group of icons if you wish. If the icons have not been written over or erased, the user will not detect that they have actually been redrawn.

The next two entries allow you to place the mouse pointer anywhere on the screen after the icons are drawn. Following these three pieces of information, *DoIcons* expects six data items for each icon.

Icon image and position data

The first **.word** contains the pointer for the bit-map data of the icon image itself. At the memory location indicated by the pointer, you must have the bit-map image coded using GEOS' compaction rules. If you are using *GeoProgrammer* you need only paste the photo scrap image of the icon into your source file. This makes it easy to use the graphic tools in *GeoPaint* to create just about any image you like without having to worry about the rather complicated compaction techniques.

Be sure you carefully follow the instructions in the *GeoProgrammer* manual. Otherwise, you'll have to break the image down manually according to the compaction formats described on page 89-90 of the *Official GEOS Programmers Reference Guide*.

The next two **.bytes** are used to locate the icon on the screen. Note that the horizontal position of the icon is restricted to every even byte. This means that you'll only be able to locate your icons to 40 different positions horizontally. This may be an important constraint in some applications. The vertical position of the icon can be on any of the 200 scan lines available.

The fifth and sixth bytes describe the size of icon. The fifth byte tells GEOS the width of the icon and byte six is the height in scanlines. As with the position of the icon, the size of the icon horizontally is a minimum of eight pixels. The minimum icon height is one scan line. The maximum icon size is eight bytes wide and 32 scanlines high. This means that you can almost completely cover the screen with adjacent icons.

Calling service routines

The final icon entry in the icon table is the **.word** pointer to the service routine to call when the icon is activated. The service routine can do almost anything but it should usually end with a JSR so that control is returned to the *DoIcons* routine. Of course, if you needed to, you could activate an icon which called a new *DoIcons*; thereby modifying, or completely redoing, the icon table.

This is probably what happens, for instance, when you click on the pattern change box in the lower left corner of the *GeoPaint* screen. A new *DoIcons* is called that uses a different icon table for the pattern icons. After a pattern is selected the original *DoIcons* is called to reactivate the toolbox. Remember, all previously drawn icon images will remain intact when a new *DoIcons* is called unless the icon table used rewrites them.

"Whoa, wait a second! If only 31 icons are allowed how come there are 32 pattern icons in *GeoPaint*?"

Icon tricks

Here is one easy way to accomplish this that I have used in my own programs. Remember that after clicking on the tool to change the pattern or after selecting the change brush menu item, the mouse is constrained to an area in the lower edge of the screen that is completely filled with adjacent icons. This is the key! Because the mouse is restricted to the selection area, you are forced to make a decision before the mouse can move out of the area. This means that one of the pattern or brush icons doesn't really have to be an icon.

Whenever the mouse button is clicked and the pointer is not over an icon or menu item, GEOS automatically calls a routine pointed to by *otherPressVector*. The routine called by *otherPressVector* is defined by the applications programmer. Upon system initialization, GEOS loads a 0 into this address and if such an event is needed it is up to the programmer to load the address of the location of the routine. It would be a relatively simple matter to write a routine pointed to by *otherPressVector* that would detect and react accordingly to the mouse button being pressed over the one pattern or brush icon that isn't really in the *DoIcons* table.

Although most applications will seldom need more than 31 icons active at one time, GEOS provides yet another way to include a few more. The routine *IsMseInRegion* tests if the pointer is inside a rectangular area of any size that you define. With this routine you're not restricted to even byte horizontal position or width. You load the vertical size and position of the region in **r2** and the horizontal size and position in **r3** and **r4** prior to calling the routine. If the pointer is inside the region when this routine is called, the accumulator will signal TRUE (-1) otherwise the accumulator will be FALSE (0). Call this routine in your otherPress routine, branch on the accumulator being TRUE and you have your own, custom, 'DoIcons' routine!

Remember the warning to always include an icon in your program? If you decide that an icon is simply not needed in your application, be sure to place an 'invisible' icon in it anyway. Simply use the following icon table:

```
;Here is the mandatory 'invisible' icon routine
JSR    DoIcons   ;Always at least one DoIcons . .
.word  DummyIcon ;Pointer to the dummy icon table.
DummyIcon:       ;Start of the dummy icon table.
.byte  1        ;Here's the single icon.
.word  160      ;wherever you want the mouse to
.byte  100      ;be after this routine is finished.
.word  IconPic ;Pointer to an empty address.
.byte  0        ;Place the blank icon in the upper
.byte  0            ;left corner of the screen.
.word  NextAddress ;Pointer to the next section.
NextAddress:
. . . . . . .          ;The rest of your code follows.
```

What are some other ways icons are used? How about the icon used to position the *GeoWrite* window on a particular section of a page? When the user clicks on this icon the service routine called probably creates a sprite with the cursor-box as its graphic data and forces it to follow the vertical position of the mouse. The mouse is constrained to moving up and down only within the limits of the box that represents the page. When the mouse is clicked again, the sprite is disabled and the new position of the window on the page is calculated and the screen moved to the proper location.

There are all kinds of interesting ways to use GEOS routines and this one in particular. Even if you've never given assembly language a try before, GEOS makes the struggle to learn it a lot easier. With the many, many service routines GEOS provides, you don't need to worry about the messy, difficult aspects of developing the user interface. You need only worry about exactly what the called service routine is going to do when its icon is clicked on.

Even if you already have a good 6502 assembler, I recommend buying Berkeley Softworks' *GeoProgrammer*. Its strong points are the sample applications they have included, the ease with which you can include graphic images without having to break them down into their compacted hexadecimal equivalents and the ability to directly create the special GEOS file header. *GeoProgrammer* produces relocatable code so it also includes a linker. And, since programs seldom run right the first time, Berkeley Softworks provides a memory resident debugger.

While my experiences with *GeoProgrammer*'s Assembler and Linker have been very positive, I'm afraid I can't say the same about the Debugger. Since memory on the 64 is limited, a mini-debugger is provided that can be loaded into the directly accessed system memory or there is a super-debugger that can be loaded into a RAM expansion unit. The super-debugger is much more powerful but, if you don't have access to an REU you can't use it.

My experience with the mini-debugger has been very frustrating. Even when following the documentation examples word-for-word, the mini-debugger would frequently lock the system up. I recommend avoiding the mini-debugger and buying or borrowing an REU if you can.

If you are going to be using *GeoProgrammer*, be aware that it does not include an editor with which to actually create your source file. I recommend using one of the more powerful versions of *GeoWrite* as your editor. While *GeoProgrammer* will not work with GEOS 128 *[See "Inside GEOS 128" in this issue. - MO]*, you can still create GEOS 128 programs. I use *Writer's Workshop 128* for editing. Besides the search and replace functions which are very useful in long listings, the 80-column screen and faster speed are great time savers. When you are finished editing, simply copy your source file to the assembler disk and re-boot your 128 in 64 mode with GEOS 64.

GeoProgrammer is not the only way to go however. I (and many other programmers) have successfully used Commodore's own *Macro Assembler Development System*, amoung other assemblers. Only Berkeley's is designed specifically for GEOS applications, however. The other C64 assembler systems will require a lot more work in developing application software.

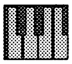
No matter what assembler you use you'll need a copy of the *Official GEOS Programmers Guide* published by Bantam. Since Berkeley Softworks is completely revising this guide, if you don't already have a copy, try to borrow one. Even if you have *GeoProgrammer* and its sample applications, try creating your own. Ideas for programs using icons are literally everywhere. Look how they are used to call *GeoPaint* tools, patterns, brush patterns, colors, etc.

Applications can use icons in very unusual ways. The assembler source listing that follows, for instance, uses icons as piano keys. When the mouse pointer is positioned over a key icon and the button pressed, the corresponding note will play. If you're interested, this keyboard could easily be expanded to include changing the waveform, volume, envelope, etc. Develop your own mouse-driven synthesizer!

I've also included the *GeoProgrammer* linker and header listings. Note that the icon I used in the *geoKeyboard.hdr* file can be substituted by the sample sequential program icon included with *GeoProgrammer*. You can then use an icon editor to edit the icon anyway you wish. If you're not going to be using *GeoProgrammer*, the listings may have to be edited to suit your own assembler. [DeskTop icons are three bytes wide and 21 scanlines high. If you're using a different assembler you can create the icon with a sprite editor and insert the data into your source as **.byte** statements. - MO]

I hope you'll now have a better understanding of how to use GEOS icons and will enjoy *GeoKeyboard*. If you've never tried assembly programming before get started! Even if you have worked with in assembly language but are unsure of how to use the routines provided in the GEOS kernal, give it a try. Don't be afraid to experiment with your own ideas. Patience and imagination are important in developing any application. Getting started is always the hardest part!

```
;*********************************************************************
;
;                          geoKeybrdHdr
;
;                   Copyright 1989  By Jim Cook
;
;*********************************************************************
;
   .if   Pass1                ;Include geosSym during assembler's
   .include geosSym           ;1st pass through the header file.
   .endif                     ;End the 'IF' clause.
   .header                    ;Flag the start of header section.
   .word 0                    ;The first two bytes are always 0.
   .byte 3                    ;Width in bytes . . .
   .byte 21                   ;. . .and height in scanlines of:
```



```
;the DeskTop icon.
```

```
    .byte $80 | USR                  ;CBM file type, with bit 7 set.
    .byte APPLICATION                ;Geos program type.
    .byte SEQUENTIAL                 ;Geos file structure type.
    .word $400                       ;Load program at this address.
    .word $3ff                       ;End address for desk accessories.
    .word $400                       ;Start program execution here.
    .byte "geoKeyboard V1.0",0,0,0,$00 ;20 characters for filename.
    .byte "James E. Cook, Jr. ",0    ;20 characters for author's name.
    .endh                            ;End of header block.


;*****************************************************************
;
;           geoKeyboard
;
;  Copyright 1989   By Jim Cook
;
;*****************************************************************
;
    .include geosSym          ;These files need to be included when assem-
    .include geosMac          ;bling, provided with GeoProgrammer package.
;
v1freqlo  =  $D400            ;These global variables were left out of the
v1freqhi  =  $D401            ;geoSym file.  If you're planning on expanding
v1pwlo    =  $D402            ;on this program you should include all of the
v1pwhi    =  $D403            ;SID and CIA registers in an .include file
v1cntrl   =  $D404            ;called geosIO.
v1attdec  =  $D405
v1susrel  =  $D406
modevol   =  $D418
cia1pra   =  $DC00
cia1prb   =  $DC01
;
    .psect                    ;Signal the start of the program.
    jsr  NewDisk              ;NewDisk is needed for early versions of GEOS.
    jsr  MouseUp              ;Activate and display the mouse pointer.
    jsr  ClearSIDRegisters    ;Clear any left over data in SID registers.
    jsr  LoadSIDRegisters     ;Load SID registers for voice one only.
;
    lda  #02                  ;Clear the screen and set up the background.
    jsr  SetPattern
    jsr  i_Rectangle
    .byte 0
    .byte 199
    .word 0
    .word 319
;
    LoadW r0,Keyboard         ;Address of icon data table.
    jsr  DoIcons              ;Display the icons and activate them.
;
    lda  #0                   ;Put mouse on geos menu item.
    LoadW r0,GeosMenu         ;Put the address of the menu table in r0.
    jsr  DoMenu               ;Display menu to quit and return to DeskTop.
    rts
;
MAIN_TOP      = 0             ;Some constants for the menu structure.
MAIN_BOT      = 15
MAIN_LFT      = 0
MAIN_RT       = 29
Y_POS_TOP_ICON = 24           ;Some constants for the icon structure.
X_POS_TOP_ICON = 5
;
Keyboard:                     ;This is the start of the icon data table.
    .byte 25                  ;Number of icons.
    .word 160                 ;x position of mouse after icons are drawn.
    .byte 100                 ;y position of mouse after icons are drawn.
;       C4 natural
    .word Natural             ;graphic data for the natural or white keys.
    .byte X_POS_TOP_ICON      ;X position of the upper left corner.
    .byte Y_POS_TOP_ICON+32   ;Y position of the upper left corner.
    .byte 2, 32               ;width in bytes and height in scanlines.
    .word DoCN4               ;routine to play C natural, fourth octave.
;       C4 sharp
    .word Sharp               ;Graphic data for the sharp or black keys.
```

```
    .byte X_POS_TOP_ICON+1
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoCS4
;       D4 natural
    .word Natural
    .byte X_POS_TOP_ICON+2
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoDN4
;       D4 sharp
    .word Sharp
    .byte X_POS_TOP_ICON+3
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoDS4
;       E4 natural
    .word Natural
    .byte X_POS_TOP_ICON+4
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoEN4
;       F4 natural
    .word Natural
    .byte X_POS_TOP_ICON+6
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoFN4
;       F4 sharp
    .word Sharp
    .byte X_POS_TOP_ICON+7
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoFS4
;       G4 natural
    .word Natural
    .byte X_POS_TOP_ICON+8
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoGN4
;       G4 sharp
    .word Sharp
    .byte X_POS_TOP_ICON+9
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoGS4;
;       A4 natural
    .word Natural
    .byte X_POS_TOP_ICON+10
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoAN4
;       A4 sharp
    .word Sharp
    .byte X_POS_TOP_ICON+11
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoAS4
;       B4 natural
    .word Natural
    .byte X_POS_TOP_ICON+12
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoBN4
;       C5 natural
    .word Natural
    .byte X_POS_TOP_ICON+14
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoCN5
;       C5 sharp
    .word Sharp
```

```
    .byte X_POS_TOP_ICON+15
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoCS5
;       D5 natural
    .word Natural
    .byte X_POS_TOP_ICON+16
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoDN5
;       D5 sharp
    .word Sharp
    .byte X_POS_TOP_ICON+17
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoDS5
;       E5 natural
    .word Natural
    .byte X_POS_TOP_ICON+18
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoEN5
;       F5 natural
    .word Natural
    .byte X_POS_TOP_ICON+20
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoFN5
;       F5 sharp
    .word Sharp
    .byte X_POS_TOP_ICON+21
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoFS5
;       G5 natural
    .word Natural
    .byte X_POS_TOP_ICON+22
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoGN5
;       G5 sharp
    .word Sharp
    .byte X_POS_TOP_ICON+23
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoGS5
;       A5 natural
    .word Natural
    .byte X_POS_TOP_ICON+24
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoAN5
;       A5 sharp
    .word Sharp
    .byte X_POS_TOP_ICON+25
    .byte Y_POS_TOP_ICON
    .byte 2, 32
    .word DoAS5
;       B5 natural
    .word Natural
    .byte X_POS_TOP_ICON+26
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
    .word DoBN5
;       C6 natural
    .word Natural
    .byte X_POS_TOP_ICON+28
    .byte Y_POS_TOP_ICON+32
    .byte 2, 32
```

```
    .word DoCN6              ;The end of the icon data table.;
;
GeosMenu:                    ;Menu table to quit the program.
    .byte MAIN_TOP           ;Put menu box in upper left corner.
    .byte MAIN_BOT
    .word MAIN_LFT
    .word MAIN_RT
    .byte HORIZONTAL|1       ;Only one menu box displayed horizontally.
;
    .word GeosText           ;Location of text for menu box.
    .byte SUB_MENU           ;Clicking on box will display a sub menu.
    .word QuitMenu           ;Location of sub menu routine.
;
GeosText:                    ;Here is the text for the box . . .
    .byte "geos",0
;
QuitMenu:                    ;The sub menu routine to quit.
    .byte MAIN_BOT           ;open the menu directly below the main menu.
    .byte MAIN_BOT+15
    .word MAIN_LFT
    .word MAIN_RT
    .byte VERTICAL|1         ;Only one sub item, opened vertically.
;
    .word QuitText           ;Location of text for sub item.
    .byte MENU_ACTION        ;Clicking on this calls EnterDeskTop . . .
    .word Quit               ;. . .routine that is located here.
;
QuitText:                    ;Here is the text for the sub item box . . .
    .byte "quit",0
;
Sharp:                               ;These are graphic data for the icons.
    .byte 224,32,1,$7F,1,$FE         ;They are simple enough to easily
Natural:                             ;compact and do not need the nice Geo-
    .byte 224,31,1,$80,1,$00,2,$FF   ;Programmer graphic assembly feature.
;


DoCN4:                       ;This is the start of the routines called by
    lda   #195               ;clicking the appropriate icons.  The values
    sta   a0L                ;to play the note desired are loaded into a
    lda   #16                ;pseudo-register.  The subroutine "Play"
    sta   a0H                ;is called to actually sound the note.
    jmp   Play               ;This is repeated for each of the 25 keys.
DoCS4:
    lda   #195
    sta   a0L
    lda   #17
    sta   a0H
    jmp   Play
DoDN4:
    lda   #209
    sta   a0L
    lda   #18
    sta   a0H
    jmp   Play
DoDS4:
    lda   #239
    sta   a0L
    lda   #19
    sta   a0H
    jmp   Play;
DoEN4:
    lda   #31
    sta   a0L
    lda   #21
    sta   a0H
    jmp   Play
DoFN4:
    lda   #96
    sta   a0L
    lda   #22
    sta   a0H
    jmp   Play
DoFS4:
    lda   #181
    sta   a0L
    lda   #23
    sta   a0H
    jmp   Play
DoGN4:
    lda   #30
    sta   a0L
    lda   #25
    sta   a0H
    jmp   Play
DoGS4:
    lda   #156
    sta   a0L
    lda   #26
    sta   a0H
    jmp   Play
DoAN4:
    lda   #49
    sta   a0L
    lda   #28
    sta   a0H
    jmp   Play
DoAS4:
    lda   #223
    sta   a0L
    lda   #29
    sta   a0H
    jmp   Play
DoBN4:
    lda   #165
    sta   a0L
    lda   #31
    sta   a0H
    jmp   Play
DoCN5:
    lda   #135
    sta   a0L
    lda   #33
    sta   a0H
    jmp   Play
DoCS5:
    lda   #134
    sta   a0L
    lda   #35
    sta   a0H
    jmp   Play
DoDN5:
    lda   #162
    sta   a0L
    lda   #37
    sta   a0H
    jmp   Play
DoDS5:
    lda   #223
    sta   a0L
    lda   #39
    sta   a0H
    jmp   Play
DoEN5:
    lda   #62
    sta   a0L
    lda   #42
    sta   a0H
    jmp   Play
```

```
DoFN5:
    lda   #193
    sta   a0L
    lda   #44
    sta   a0H
    jmp   Play
DoFS5:
    lda   #107
    sta   a0L
    lda   #47
    sta   a0H
    jmp   Play
DoGN5:
    lda   #60
    sta   a0L
    lda   #50
    sta   a0H
    jmp   Play
DoGS5:
    lda   #57
    sta   a0L
    lda   #53
    sta   a0H
    jmp   Play
DoAN5:
    lda   #99
    sta   a0L
    lda   #56
    sta   a0H
    jmp   Play
DoAS5:
    lda   #190
    sta   a0L
    lda   #59
    sta   a0H
    jmp   Play
DoBN5:
    lda   #75
    sta   a0L
    lda   #63
    sta   a0H
    jmp   Play
DoCN6:
    lda   #15
    sta   a0L
    lda   #67
    sta   a0H
    jmp   Play
Play:
    jsr   InitForIO         ;Must be called to access SID or CIA.
    lda   #$40
    sta   v1cntrl           ;Make sure voice one is off.
    lda   a0L
    sta   v1freqlo          ;Put the frequency data values in the SID.
    lda   a0H
    sta   v1freqhi
    lda   #$41              ;Play the note!
    sta   v1cntrl
    ldy   #$40
20$
    ldx   #$ff              ;A loop to give the note a minimum duration.
10$                         ;Note the use of local labels 10$ and 20$.
    nop                     ;Read more about them in the geoProgrammer
    nop                     ;manual.
    dex
    bne   10$
    dey
    bne   20$
30$
    LoadB cia1pra,%11111111 ;Disable keyboard matrix columns so only the
    lda   cia1prb           ;mouse button is examined.  Use exclusive or
    eor   #$FF              ;to complement data.  To mask out the
    and   #%00010000        ;other inputs, AND a one on the data line
    cmp   #%00010000        ;for the button input and compare . . .
    beq   30$               ; . . . branch if button is still being held.
    lda   #$40              ;Mouse button released. . . stop playing!
    sta   v1cntrl
    jsr   DoneWithIO        ;Restore standard GEOS configuration.
    rts                     ;Return to looking at the icons.;
ClearSIDRegisters:
    jsr   InitForIO         ;This routine is used to flush out any old
    lda   #$0               ;data stored in all the SID registers.
    ldx   #$18              ;loop to zero each register and executes a
10$                         ;'DoneWithIO jsr when done looping.
    sta   v1freqlo,X
    dex
    bne   10$
    jsr   DoneWithIO
    rts
;
LoadSIDRegisters:
    jsr   InitForIO         ;Must be called in to access SID and CIA's.
    lda   #$40
    sta   v1cntrl           ;Make certain voice is off.
    lda   #$0f
    sta   modevol           ;Set volume full.
    lda   #$09
    sta   v1attdec          ;Set attack and decay to piano envelope.
    lda   #$01
    sta   v1susrel          ;Set sustain and release to piano envelope.
    lda   #15
    sta   v1pwhi            ;Set pulse width registers to piano waveform.
    lda   #36
    sta   v1pwlo
    jsr   DoneWithIO        ;Finished with I/O operations return to GEOS.
    rts
;
Quit:
    jmp   EnterDeskTop      ;Leave geoKeyboard and return to the DeskTop.


;*************************************************************************
;
;                       geoKeyboard.lnk
;
;                  Copyright 1989  By Jim Cook
;
;*************************************************************************
;
    .output geoKeyboard     ;Name for sequential output file.
    .header geoKeybrdHdr.rel ;Name of file containing header block.
    .seq                    ;Flag Linker, this is a sequential program.
    .psect  $0400           ;Program code starts at address $0400.

geoKeyboard.rel             ;The name of the relocatable file which
                            ;contains code and data created by Geo-
                            ;Assembler.  Note that GeoAssembler automatic-
                            ;ally appends a .rel to your source file name.
```

BASIC 2.0 Array Shell Sort

Putting arrays together and taking sorts apart

by Anton Treuenfels

There is no single 'best' sort for all imaginable sorting problems, so different sort routines designed for different problems will strike different balances among such competing requirements as size, speed, versatility, and robustness. The sort routine described here is an example of one such balance. It is less than 512 bytes long, can sort 1000 randomly ordered strings in less than seven seconds, can sort any BASIC 2.0 singly-dimensioned array in either ascending or descending order, and tries to behave reasonably in the face of what it considers to be errors.

Using the program

The program is assembled to run in the popular free RAM block starting at $C000, although it can of course be reassembled to run somewhere else (it might even be modified to become a *TransBasic* module). It can be LOADed using the **,8,1** syntax in either immediate or program mode.

Sorts are invoked with a SYS call:

sys 49152, arynam(e1), arynam(e2)

where 49152 is the start address, **arynam** is the name of the array to sort, and **e1** and **e2** are elements of **arynam**. **Arynam** can be any legal array name (string, real, or integer). **E1** and **e2** indicate the elements that bound the sort: it is not necessary to sort the entire array if that is not desired. If **e1** is less than **e2**, the array will be sorted in ascending order; if **e1** is greater than **e2**, the array will be sorted in descending order; and if **e1** equals **e2** the sort routine exits without affecting anything.

Program performance

The program *Shellsort Test* loads the sort program and puts it through its paces. The parameters of a test are: type of array to sort, initial number of elements in the array, final number of elements, sort direction, and number of passes to average. Each 'pass' is a series of sorts starting with an array containing the specified initial number of random elements. After the array has been sorted once, the sort is executed a second time on the now-ordered array. The series continues by doubling the number of elements in the array until it is greater than the specified final number of elements. When a pass is complete another begins, until the specified number of passes is completed. Averaged results for all passes are then displayed.

There is also an option to display the array elements during the first pass. This is handy in verifying that the sort behaves as expected (which it did not always do during development). Elements are displayed in groups of twenty, four across and five down (a minor deception - strings are displayed to a maximum length of nine rather than their true maximum length of ten for a cleaner display). About sixty array elements can be comfortably displayed at once, so values up to 60 for 'number of elements to sort' work well. If more elements are used, the display can be started and stopped by pressing any key.

The test program reported these average results (in seconds):

#Elements	String	Real	Integer
125	0.47/0.26	0.55/0.28	0.34/0.18
250	1.15/0.60	1.36/0.65	0.82/0.43
500	2.77/1.37	3.23/1.50	1.99/0.97
1000	6.53/3.13	7.67/3.40	4.64/2.19

10-pass averages (random/ordered)

About the program

The calling syntax of **name(element one), name(element two)** is designed to solve two problems. First, it helps to guarantee that the sort routine has two valid pointers into the same array. A common alternative syntax is **name(element), #elements**, which makes it possible that **#elements** might accidentally be larger than the actual number of elements in the array. Without expensive error-checking, the sort routine might happily go on to sort memory that wasn't actually part of the array, leading to all kinds of nasty side effects. The syntax chosen makes it fairly cheap in terms of time and code to verify that the pointers are 'reasonable', although it does not go so far as to guarantee that the array is singly-dimensioned.

The other problem is how to flag which way to order the array. It is certainly easy to require a third parameter, but it is also easy to just let the sort routine compare the two pointers and decide for itself.

The heart of a sort routine can be regarded as a group of four basic tasks repeatedly executed in a loop structure: *decide* which two objects to compare, *find* them, *compare* them and, if necessary, *exchange* them (or the things used to find them). The main difference between various sort algorithms is the method of deciding which two objects to compare. On the other hand, the main differences in sorting different objects are how they are found, compared and exchanged.

The program presented here divides the four basic tasks into a single Decide routine for all objects and a separate Find, Compare and Exchange routine for each different object. Although in the interest of saving space there are several places in the code where sharing or overlap occurs, in principle there are ten separate routines (1 Decide, 3 Find, 3 Compare, 3 Exchange) and there is nothing to prevent any one of them from being replaced independently of any other (to gain speed at the expense of size, for example). The possibility of adding routines to sort other objects or implementing other sort algorithms altogether is also open.

The Decide routine (the actual sort algorithm) is a *Shell* sort. It is a reasonably compact, fast, and understandable algorithm that performs well on random and even better on ordered collections of objects. Alternative algorithms that might be used here are the *Insertion* sort (about the same code size; performs best on nearly ordered collections) and the *Quick* sort (larger code size; as usually implemented performs best on random collections).

The Find routines locate the objects to be compared, taking into account object sizes and the fact that, while the contents of real and integer arrays are reals and integers, the contents of string arrays are not strings but instead string descriptors. The separateness of the Find step is an often overlooked part of a sort routine. In BASIC 2.0, for example, a variable is automatically located whenever it occurs in the program text, so that in a BASIC 2.0 sort routine what looks like a Compare operation is really a combined Find and Compare operation.

The Compare routines assume that, given any two objects of the same type, the statement can always be made that in some sense the first is less than, equal to or greater than the second object. The Compare routines determine which is the case and return a flag byte in which set bits represent logical relationships that are TRUE. For example, if the first object is less than the second, then all of the logical relations 'less', 'less or equal' and 'not equal' are true.

This system of return values from the Compare routines is a little more complex than that used by compare routines in many other sorts (often simply a negative value for less than, zero for equal, positive for greater). The advantage is that the same code can sort in either direction by setting a few flags at the start of the Decide routine, rather than having to write a separate routine for each direction. For example, the Decide routine of *Shellsort* calls the Compare routines looking for either 'greater or equal' or 'less or equal' (depending on which direction the sort runs), and exchanges elements if the desired relation is NOT TRUE.

The Exchange routines are straightforward. It is interesting to note that, even though the process of locating strings requires a level of indirection, exchange is similar to that of reals and integers. Of course, it is the string descriptors rather than the strings themselves that are being exchanged.

The idea of sorting descriptors of objects instead of the objects themselves can be taken more generally. For example, integers in one array can be treated as representing element numbers of a second array. A keysort arranges the integers in the first array according to the values in the second, so that the first array becomes an 'index' to the second. It would require only a slight modification of the Find and Compare routines of *Shellsort* to implement a keysort of any array type. It would even be handy: there are often occasions where it is more useful to have a sorted index to an array than a sorted array.

In effect, any of the various modifications that might be made to extend or change *Shellsort* amount to striking another of the many different possible balances among sort requirements. Readers are welcome to use any of the material presented here in their own efforts to strike useful balances.

Listing 1: *shellsort.s - Merlin format*

```
* basic 2.0 array shell sort
* last revision:  09/28/88

* written by anton treuenfels
* 5248 horizon drive
* fridley, minnesota usa  55421
* 612/572-8229

* program constants

asmadr = $c000        ;assembly address

tstne  = %100000      ;not equal
tstgt  = %010000      ;greater
tstge  = %001000      ;greater or equal
tsteq  = %000100      ;equal
tstle  = %000010      ;less or equal
tstls  = %000001      ;less

isgrt  = tstne.tstgt.tstge
isequ  = tstge.tsteq.tstle
isles  = tstne.tstle.tstls

* program zero-page usage

e1ptr  = $22          ;pointer -> 1st element
e2ptr  = $24          ;pointer -> 2nd element
elm1   = $26          ;1st element index
elm2   = $28          ;2nd element index
```

```
elmcnt = $58         ;#elements
produc = $5a         ;multiplication product

e1len  = $60         ;1st string descriptor
e1adr  = $61
e2len  = $63         ;2nd string descriptor
e2adr  = $64

tstflg = $69         ;test-type flag
delta  = $6a         ;comparison distance
lstelm = $6c         ;last element this pass
crrelm = $6e         ;current element this pass

* basic 2.0 zero-page usage

arysta = $2f         ;start-of-arrays
varnam = $45         ;current variable name
varadr = $47         ;current variable address

* basic 2.0 indirect vectors

baserr = $300        ;error report

* basic 2.0 rom

chkcom = $aefd       ;check and skip comma
fndvar = $b08b       ;locate variable
memfac = $bba2       ;transfer memory to fac
cmpfac = $bc5b       ;compare memory to fac

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

 org asmadr

* basic 2.0 interface

       jsr chkcom    ;get 1st parameter
       jsr fndvar
       ldx #0
       jsr savptr    ;save variable address
       lda varnam+1
       pha           ;save variable name
       lda varnam
       pha
       jsr chkcom    ;get 2nd parameter
       jsr fndvar
       ldx #2
       jsr savptr
       ldx #8*3      ;assume string
       pla           ;1st char of 1st name
       bpl par1      ;b:real or string
       ldx #8*1      ;integer
par1   eor varnam    ;names match?
       bne typerr    ;b:no
       pla           ;2nd char of 1st name
       bmi par2      ;b:string or integer
       ldx #8*2      ;real
par2   eor varnam+1
       bne typerr
       ldy #8*1
]bb1   lda isort-1,x ;set remaining parameters
       sta elmsiz-1,y
       dex
       dey
       bne ]bb1
       jsr arysrt    ;execute sort
       rts

* save pointer to element

savptr lda varadr
       sta begelm,x  ;save location
       cmp arysta    ;verify array element
       lda varadr+1
       sta begelm+1,x
       sbc arysta+1
       bcc typerr    ;b:not array element
       rts

* report error

typerr ldx #22 ;'type mismatch'
       jmp (baserr)

* sort parameter tables

isort  da  2         ;#bytes/element
       da  fndint    ;find routine
       da  cmpint    ;compare routine
       da  excint    ;exchange routine

fsort  da  5
       da  fndflp
       da  cmpflp
       da  excflp

ssort  da  3
       da  fndstr
       da  cmpstr
       da  excstr

* array sort parameter format:

* word begelm - location of first element
* word endelm - location of last element
* word elmsiz - #bytes in an element
* word fndadr - location of find routine
* word cmpadr - location of comparison routine
* word swpadr - location of exchange routine

* array sort

arysrt ldx begelm
       ldy begelm+1
       cpy endelm+1
       bcc ays2      ;b:first<last
       bne ays1
       cpx endelm
       bcc ays2
       beq ays3      ;b:first=last (one element)
ays1   lda endelm    ;exchange pointers
       sta begelm
       lda endelm+1
       sta begelm+1
       stx endelm
       sty endelm+1
ays2   php           ;save direction flag
       jsr fndcnt    ;get element count
       plp
       jsr shlsrt    ;execute sort
ays3   rts

* determine element count

fndcnt sec
       lda endelm    ;#bytes in array
       sbc begelm
       sta elmcnt
       lda endelm+1
       sbc begelm+1
       sta elmcnt+1
       ldy #16
       lda #0
```

```
]bb1   asl elmcnt    ;divide by #bytes/element
       rol elmcnt+1
       rol
       cmp elmsiz
       bcc fdc1
       sbc elmsiz
       inc elmcnt    ;result in elmcnt
fdc1   dey
       bne ]bb1
       rts

* shell sort

shlsrt lda elmcnt
       sta delta     ;delta= elmcnt
       lda elmcnt+1
       sta delta+1
       lda #tstle    ;ascending sort
       bcc shl1
       lda #tstge    ;descending sort
shl1   sta tstflg    ;test-result flag
       bne shl4      ;enter outer loop

* outer loop

]bb1   sec
       lda elmcnt    ;lstelm= elmcnt-delta
       sbc delta
       sta lstelm
       lda elmcnt+1
       sbc delta+1
       sta lstelm+1
       lda #0
       sta crrelm    ;crrelm= 0
       sta crrelm+1

* middle loop

]bb2   clc
       lda crrelm
       sta elm1      ;elm1= crrelm
       adc delta
       sta elm2      ;elm2= crrelm+delta
       lda crrelm+1
       sta elm1+1
       adc delta+1
       sta elm2+1

* inner loop

]bb3   jsr fndelm    ;find elements
       jsr compar    ;compare elements
       and tstflg    ;in proper order?
       bne shl2      ;b:yes
       jsr exchng    ;swap elements
       sec
       lda elm1
       sta elm2      ;elm2= elm1
       sbc delta
       sta elm1      ;elm1= elm1-delta
       lda elm1+1
       sta elm2+1
       sbc delta+1
       sta elm1+1
       bcs ]bb3      ;b:elm1 >= 0

* inner loop end

shl2   inc crrelm    ;crrelm= crrelm+1
       bne shl3
       inc crrelm+1
shl3   lda lstelm
       cmp crrelm
       lda lstelm+1
       sbc crrelm+1
       bcs ]bb2      ;b:lstelm >= crrelm

* middle loop end

shl4   lsr delta+1   ;delta= int(delta/2)
       ror delta
       lda delta+1   ;delta= 0?
       ora delta
       bne ]bb1      ;b:no - continue

* outer loop end

       rts           ;finished

* find two elements

fndelm jmp (fndadr)  ;to current handler

fndint ldy #2-1
       dfb $2c
fndflp ldy #5-1
       dfb $2c
fndstr ldy #3-1
       ldx #2
]bb1   lda elm1+1,x
       sta produc+1
       lda elm1,x
       cpy #3-1      ;integer?
       bcc fnd1      ;b:yes
       asl           ;*2
       rol produc+1
       cpy #5-1      ;string?
       bcc fnd1      ;b:yes
       asl           ;*4
       rol produc+1
fnd1   adc elm1,x    ;*2,*3,*5
       sta produc
       lda produc+1
       adc elm1+1,x
       sta produc+1
       lda produc
       adc begelm    ;add offset to base address
       sta elptr,x
       lda produc+1
       adc begelm+1
       sta elptr+1,x
       dex
       dex
       beq ]bb1      ;b:do second element
       cpy #3-1      ;string?
       bne fnd2      ;b:no
]bb2   lda (elptr),y ;get string descriptors
       sta e1len,y
       lda (e2ptr),y
       sta e2len,y
       dey
       bpl ]bb2
fnd2   rts

* compare two elements

compar jmp (cmpadr)  ;to current handler

* compare integers

cmpint ldy #0        ;most significant byte
       sec
       lda (elptr),y
```

```
        sbc (e2ptr),y
        beq cin2
        bmi cin1      ;signed compare
        bvc e1grt
        bvs e1les

cin1    bvc e1les
        bvs e1grt

cin2    iny
        lda (e1ptr),y
        sbc (e2ptr),y
        beq e1equ
        bcc e1les     ;unsigned compare

* return e1 greater than e2

e1grt   lda #isgrt
        rts

* compare floating points

cmpflp jsr memfac+4  ;e1 to fac
        jsr cmpfac+4  ;compare e2 to fac
        bmi e1les     ;b:e1<e2
        bne e1grt     ;b:e1>e2

* return e1 equal e2

e1equ   lda #isequ
        rts

* compare strings

cmpstr lda e1len
        cmp e2len     ;compare lengths
        bcc cst1      ;b:e1<e2
        lda e2len
cst1    tax           ;use shorter string
        beq cst2      ;b:shorter is null
        ldy #-1
]bb1    iny
        lda (e1adr),y ;chars different?
        cmp (e2adr),y
        bne cst3      ;b:yes
        dex           ;all chars compared?
        bne ]bb1      ;b:no
cst2    lda e1len     ;lengths different?
        cmp e2len
        beq e1equ     ;b:e1=e2
cst3    bcs e1grt     ;b:e1>e2

* return e1 less than e2

e1les   lda #isles
        rts

* exchange elements

exchng jmp (swpadr)  ;to current handler

excint ldy #2-1
        dfb $2c       ;skip next two bytes
excflp ldy #5-1
        dfb $2c
excstr ldy #3-1
]bb1    lda (e1ptr),y
        tax
        lda (e2ptr),y
        sta (e1ptr),y
        txa
        sta (e2ptr),y
        dey
        bpl ]bb1
        rts

* sort parameter storage

 dum *+1&$fffe        ;word-align

begelm ds  2  ;pointer -> first element
endelm ds  2  ;pointer -> last element
elmsiz ds  2  ;#bytes/element
fndadr ds  2  ;vector -> find routine
cmpadr ds  2  ;vector -> compare routine
swpadr ds  2  ;vector -> exchange routine

 dend
```

Listing 2: *BASIC generator for "shellsort.o"*

```
BD  100 rem  generator for "shellsort.o"
GC  110 n$="shellsort.o": rem  name of program
FE  120 nd=495: sa=49152: ch=61786

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

LB  1000 data  32, 253, 174,  32, 139, 176, 162,   0
BB  1010 data  32,  64, 192, 165,  70,  72, 165,  69
IG  1020 data  72,  32, 253, 174,  32, 139, 176, 162
LE  1030 data   2,  32,  64, 192, 162,  24, 104,  16
KK  1040 data   2, 162,   8,  69,  69, 208,  42, 104
HM  1050 data  48,   2, 162,  16,  69,  70, 208,  33
IK  1060 data 160,   8, 189,  85, 192, 153, 243, 193
OF  1070 data 202, 136, 208, 246,  32, 110, 192,  96
DN  1080 data 165,  71, 157, 240, 193, 197,  47, 165
PL  1090 data  72, 157, 241, 193, 229,  48, 144,   1
HH  1100 data  96, 162,  22, 108,   0,   3,   2,   0
OL  1110 data  57, 193, 136, 193, 217, 193,   5,   0
PJ  1120 data  60, 193, 167, 193, 220, 193,   3,   0
EM  1130 data  63, 193, 180, 193, 223, 193, 174, 240
CN  1140 data 193, 172, 241, 193, 204, 243, 193, 144
GK  1150 data  27, 208,   7, 236, 242, 193, 144,  20
BM  1160 data 240,  26, 173, 242, 193, 141, 240, 193
HO  1170 data 173, 243, 193, 141, 241, 193, 142, 242
BE  1180 data 193, 140, 243, 193,   8,  32, 157, 192
ON  1190 data  40,  32, 197, 192,  96,  56, 173, 242
CA  1200 data 193, 237, 240, 193, 133,  88, 173, 243
KD  1210 data 193, 237, 241, 193, 133,  89, 160,  16
JB  1220 data 169,   0,   6,  88,  38,  89,  42, 205
NO  1230 data 244, 193, 144,   5, 237, 244, 193, 230
DN  1240 data  88, 136, 208, 238,  96, 165,  88, 133
CM  1250 data 106, 165,  89, 133, 107, 169,   2, 144
AC  1260 data   2, 169,   8, 133, 105, 208,  84,  56
IL  1270 data 165,  88, 229, 106, 133, 108, 165,  89
EO  1280 data 229, 107, 133, 109, 169,   0, 133, 110
GC  1290 data 133, 111,  24, 165, 110, 133,  38, 101
OC  1300 data 106, 133,  40, 165, 111, 133,  39, 101
IB  1310 data 107, 133,  41,  32,  54, 193,  32, 133
AN  1320 data 193,  37, 105, 208,  22,  32, 214, 193
GM  1330 data  56, 165,  38, 133,  40, 229, 106, 133
JN  1340 data  38, 165,  39, 133,  41, 229, 107, 133
DA  1350 data  39, 176, 224, 230, 110, 208,   2, 230
JG  1360 data 111, 165, 108, 197, 110, 165, 109, 229
MH  1370 data 111, 176, 191,  70, 107, 102, 106, 165
EM  1380 data 107,   5, 106, 208, 162,  96, 108, 246
HI  1390 data 193, 160,   1,  44, 160,   4,  44, 160
CL  1400 data   2, 162,   2, 181,  39, 133,  91, 181
OB  1410 data  38, 192,   2, 144,  10,  10,  38,  91
OA  1420 data 192,   4, 144,   3,  10,  38,  91, 117
HM  1430 data  38, 133,  90, 165,  91, 117,  39, 133
ID  1440 data  91, 165,  90, 109, 240, 193, 149,  34
FC  1450 data 165,  91, 109, 241, 193, 149,  35, 202
DC  1460 data 202, 240, 208, 192,   2, 208,  13, 177
PE  1470 data  34, 153,  96,   0, 177,  36, 153,  99
```

```
DP 1480 data   0, 136,  16, 243,  96, 108, 248, 193
BM 1490 data 160,   0,  56, 177,  34, 241,  36, 240
FP 1500 data  10,  48,   4,  80,  15, 112,  60,  80
HP 1510 data  58, 112,   9, 200, 177,  34, 241,  36
OB 1520 data 240,  15, 144,  47, 169,  56,  96,  32
CF 1530 data 166, 187,  32,  95, 188,  48,  36, 208
LG 1540 data 243, 169,  14,  96, 165,  96, 197,  99
MA 1550 data 144,   2, 165,  99, 170, 240,  12, 160
IP 1560 data 255, 200, 177,  97, 209, 100, 208,   9
FJ 1570 data 202, 208, 246, 165,  96, 197,  99, 240
OG 1580 data 224, 176, 209, 169,  35,  96, 108, 250
PE 1590 data 193, 160,   1,  44, 160,   4,  44, 160
OB 1600 data   2, 177,  34, 170, 177,  36, 145,  34
HA 1610 data 138, 145,  36, 136,  16, 243,  96
```

Listing 3: *Shellsort Test*

```
DB 100 print"{down}* array shell sort tester"
DC 105 print"* by anton treuenfels"
LF 110 print"* last revised - 09/28/88"
HO 115 :
FE 120 ifpeek(49152)<>32orpeek(49153)<>253thena=peek(186):load"shellsort.o",a,1
BP 125 :
MN 150 me=1000:sz=20:mp=20
AJ 155 im=32000:ir=16000
KN 160 rm=1000000:rr=500000:rp=100
JB 165 :
IJ 180 print"{down}define test parameters:"
NC 185 :
NG 200 p$="array type (string, integer, real)":q$="sir":gosub975:at=a
BE 205 :
MO 210 p$="{down}initial #elements":p=1:q=me:r=int(me/8):gosub950:ie=a
LE 215 :
LI 220 p$="{down}final #elements":p=ie:q=me:r=me:gosub950:fe=a
FF 225 :
CK 230 p$="{down}sort order (ascending, descending)":q$="ad":gosub975:so=a
PF 235 :
IE 240 p$="{down}#passes to average":p=1:q=200:r=1:gosub950:np=a
JG 245 :
AN 250 p$="{down}display first pass (no, yes)":q$="ny":gosub975:df=a
DH 255 :
OD 300 deffnr0(a)=int(rnd(1)*a)
GA 305 deffnr1(a)=int(rnd(1)*a)+1
KJ 310 deffnrn(a)=int((a+.005)*100)/100
PK 315 :
HN 330 ifat=1thendimar$(me)
BO 335 ifat=2thendimar%(me)
KM 340 ifat=3thendimar(me)
NM 345 :
JM 350 dimup(mp),sp(mp)
HN 355 :
GB 400 forpc=1tonp
DC 405 print"{down}pass#";pc
OA 410 :
II 420 cc=ie:ci=1
AE 425 ifat=1thengosub750
CC 430 :
EH 450 gosub800
LD 455 p$="random="
IH 460 gosub600
JK 465 up(ci)=up(ci)+(et-up(ci))/pc
HG 470 p$="sorted="
HI 475 gosub600
KK 480 sp(ci)=sp(ci)+(et-sp(ci))/pc
JF 485 :
BJ 500 ifat=1thengosub780
JG 505 cc=2*cc:ci=ci+1
EF 510 ifcc<=fethen450
HH 515 :
IA 520 nextpc
BI 525 :
HG 530 print"{down}{down}averages over";np;"passes:"
IE 535 print"{down}#elements","random","sorted"
CD 540 print:cc=ie:ci=1
IB 545 printcc,fnrn(up(ci)),fnrn(sp(ci))
GJ 550 cc=2*cc:ci=ci+1
EJ 555 ifcc<=fethen545
EK 560 :
CA 565 p$="{down}another test (no, yes)":q$="ny":gosub975:ifa=2thenclr:goto180
OK 570 :
EE 580 end
NL 585 :
CJ 600 printcc;p$;
AI 605 ifso=1thena=1:b=cc
ON 610 ifso=2thena=cc:b=1
KN 615 onatgoto620,625,630
MI 620 c=ti:sys49152,ar$(a),ar$(b):d=ti:goto635
IJ 625 c=ti:sys49152,ar%(a),ar%(b):d=ti:goto635
HK 630 c=ti:sys49152,ar(a),ar(b):d=ti
HG 635 et=fnrn((d-c)/60)
PM 640 printet;"seconds"
EB 645 ifdf=2andpc=1thengosub850
GK 650 return
DA 655 :
LA 750 print"{down}generating master string..."
HP 755 a=rnd(rnd(-ti)):ms$=""
BG 760 fori=1to255:ms$=ms$+chr$(fnr0(26)+65):next:return
BH 765 :
FI 780 fori=1tocc:ar$(i)="":next:a=fre(0):return
FI 785 :
EJ 800 print"{down}generating";cc;"elements..."
AH 805 a=rnd(rnd(-ti))
HL 810 onatgoto815,820,825
FI 815 a=8+df:fori=1tocc:ar$(i)=mid$(ms$,fnr1(240),fnr1(a)):next:goto830
DI 820 fori=1tocc:ar%(i)=fnr1(im)-ir:next:goto830
IC 825 fori=1tocc:ar(i)=(fnr0(rm)-rr)/rp:next
NM 830 ifdf=2andpc=1thengosub850
PF 835 return
ML 840 :
CP 850 fori=0tocc-1stepsz
EH 855 print:onatgoto860,865,870
AH 860 forj=1tosz:printleft$(ar$(i+j),9),:next:goto875
MG 865 forj=1tosz:printar%(i+j),:next:goto875
GH 870 forj=1tosz:printar(i+j),:next
AC 875 gosub900
EH 880 next
OD 885 print:print"{down}press any key{down}":gosub905
GJ 890 return
DP 895 :
PN 900 geta$:ifa$=""then910
BP 905 geta$:ifa$=""then905
KK 910 return
HA 915 :
DF 950 a$=str$(p):b$=str$(q):c$=str$(r)
MF 955 printp$;" (";a$;" -";b$;") ";c$;left$("{7 left}",len(c$)+1);
BB 960 inputa:a=int(a):ifa<pora>qthen955
BO 965 return
OD 970 :
BC 975 printp$;"  ";left$(q$,1);"{left}{left}{left}";
FJ 980 inputa$:a$=left$(a$,1)
DN 985 fora=1tolen(q$):ifa$=mid$(q$,a,1)then995
KP 990 next:goto975
PP 995 return
```

A *glob* Function For Power C

Wildcard pattern matching

by Adrian Pepper

The Power C shell provides a pleasing command-line interface. However, I often found myself wishing it would provide the useful 'wildcard' file name facilities of systems such as MS-DOS and UNIX.

'Wildcard' is a term used to refer to the ability to specify many file names at once in a command line by having the command line interpreter treat some arguments not just as single literal names, but as patterns matching sets of file names. Many of us get used to typing `*.c` on other systems to match all file names ending with **.c**. Here the * is not interpreted literally, but is a *metacharacter*, used to mean "any number of repetitions of any character". From the list of all currently available file names, all matching file names are selected and the command functions as if all the names had been literally typed in place of the pattern.

For reasons buried deep in the fifteen-year antiquity of UNIX development, programs and routines for expanding wildcard patterns into lists of matching file names often go by the name *glob*. It is an abbreviation of "global", and a reference to the fact that the command will be executed "globally"; that is to say, for all appropriate available file names. A very narrow-minded use of the word "global", if you think about it.

Although the Power C shell does not provide this, it proved not too difficult to create a generalized means to allow Power C programs to easily support it themselves.

This article attempts to not only give a solution, but also to trace the steps by which a concept can become a practical tool. In addition, it attempts to give a little insight into the C language and the Power C package.

Strangely enough, though, the algorithms and system details presented can be easily adapted to a variety of languages.

Reading a disk drive directory with Power C

The first step was to verify that Power C programs could quite easily process directory listings from 1541s and similar disk drives. This directory listing would be considered our list of available file names for pattern matching. The simple *dir* program illustrated does this. It essentially mimics the built-in shell **l** command which gives a directory listing of the drive currently designated as the 'work' drive.

```
/*
 * dir.c - basic-style directory listing
 */
#include <stdio.h>

static char buf[100];
main(argc,argv)
unsigned argc;
char **argv;
{
  char *pat;
  unsigned dev;
  FILE fid;
  unsigned n;

  /* command line pattern if given, else empty string */
  pat = (argc > 1) ? argv[1] : "";
  sprintf(buf,"$%s", pat);
#define wrkdev (*(char *)0x17fc)
  dev = wrkdev;  /* device set by shell work command */

  fid = 5;
  if (!open(fid, dev, 0, buf) _
    ferror() ) {
    printf("Can't open %s on device %d\n", buf, dev);
    exit(1);
  }

  fgetc(fid);
  fgetc(fid);  /* skip "load address" */

  while ((n = gdirline(buf,fid)) != EOF)
    printf("%u %s\n", n, buf);
  fclose(fid);  /* fclose for open() allows re-use */
}
```

The *gdirline* routine is crucial to the *dir* program:

```
/*
 * gdirline.c - read a line from a
 *   directory "load" into given buf
 *
 * return as function value
 *   - the "line number" part of basic
 *     style line (that is, number of
 *     blocks)
 *   - EOF at end-of-file
 */
#include <stdio.h>
```

```
gdirline(buf, fid)
char *buf;
FILE fid;
{
  char *b;
  unsigned c, n;

  fgetc(fid);       /* skip "link" */
  fgetc(fid);       /*  ... */

  n = fgetc(fid);  /* get "line number", low byte */
  c = fgetc(fid);  /* and high byte */
  n += c<<8;        /* and put the two together */

  /* read rest of line; ended normally by a zero byte */
  for (b = buf; (c = fgetc(fid)) && c != EOF; ++b)
    *b = c;

  *b = '\0'; /* just in case didn't end with zero */

  if (c == EOF) return EOF;

  return n;  /* return "line number" */
             /* assume EOF is invalid line number */
}
```

The *dir* program takes advantage of a feature of the 1541 and similar disk drives by opening a 'directory listing', signified by a file name beginning with a dollar-sign character ($), but also specifying zero as the secondary address for the drive. This is interpreted specially by these drives, causing them to format and send back what appears to be a BASIC program, complete with load address, and meaningless statement links.

The number of blocks for each file is sent back as the line number for the 'statement', while the statement itself forms the rest of the directory line. Whenever the file name used for an open begins with a dollar-sign, the directory is opened. Only if a secondary address of zero is used will the special 'BASIC program' format be transmitted.

All this is why **load "$",8** in BASIC replaces your current program with something that, when listed, gives you information about the directory of the disk.

While Power C does provide library routines for reading the actual disk directory entries, there were a number of reasons for not wanting to use them. The most important was that they would not work at all with the normal RAMdisk software provided by Commodore for their 1764/1750 RAM expander, while the opening of a directory listing on secondary address zero would.

Saving a directory listing for later use

The Commodore disk drives perform limited wildcard matching when returning directory listings. A question mark (?) can be used to match any single arbitrary character, and an asterisk (*) can be used to match any number of any characters.

A preliminary routine for providing the glob facility then can be simply written, as follows:

```
/*
 * devglob.c - expand wildcard filenames
 *
 * devglob executes the given function
 *  for each file name returned by a
 *  directory listing of the given
 *  filename wildcard pattern on the
 *  given device
 *
 * returns as function value the number
 *  of names matched
 *
 * The 'select' function argument allows
 *  the file selection process to be
 *  customized.
 */
#include <stdio.h>

/*
 * this structure is to create a linked
 *  list of matching file names
 */
typedef struct link {
  struct link *l$next;  /* point to next in list */
  char *l$name;     /* point to name of this file */
} LINK;

static LINK *namelist;
static char buf[100];

devglob(dev,pat,func,select)
unsigned dev;     /* device to use for dir */
char *pat;        /* filename pattern to expand */
int (*func)();    /* function to call on matches */
int (*select)(); /* extra matcher */
{
  extern char *malloc();
  extern char *strdup();
  LINK **plink; /* keep track of end of list */
  LINK *nlink;  /* pointer to new LINK created */
  char *nname;  /* pointer to new saved name */
  LINK *llink;  /* general pointer to LINK */
  char *b;
  FILE fid;
  int n;

  sprintf(buf,"$%s", pat);    /* form directory name */

  fid = 5;
  if (!open(fid, dev, 0, buf) _ ferror() )
    return 0;

  plink = &namelist;  /* matching file names will */
  *plink = NULL;      /* be saved in a linked list */

  fgetc(fid);         /* skip "load address" */
  fgetc(fid);         /* ... */

  while (gdirline(buf,fid) != EOF) {
    if (*buf != ' ')
      ;  /* not a filename line; skip it */
    else {
      for (b = buf+strlen(buf);
        b > buf && *b != '"'; --b)
        ;
      *b = '\0';
      for (b = buf; *b != '\0' && *b++ != '"'; )
        ;  /* skip to EOS or after first '"' */
      if (!select _ (*select)(b)) {
        /* got a good one! */
        /* use malloc to create a LINK struct for list */
        nlink = (LINK *)malloc(1,sizeof(LINK));
        nname = strdup(b);      /* save copy of name */
/* note: bug in POWER C causes low byte */
/* only to be tested if assignment done inside if */
        if (!nlink _ !nname) {
```

```
            printf("glob: too many names at %s\n", b );
            break;                 /* finish prematurely */
          }
          nlink->l$name = nname;  /* point to name */
          *plink = nlink;         /* and link new LINK */
          plink = &nlink->l$next; /* link next one here */
          *plink = NULL;          /* NULL (end) for now */
        }
      }
    }
    fclose(fid);  /* fclose for open() allows re-use */

    /*
     * now run through the list of saved
     * names, executing the function for
     * each one
     */
    n = 0;
    for (llink = namelist; llink; llink=nlink) {
      if (func)
        (*func)(llink->l$name); /* call function */
      nlink = llink->l$next;    /* remember next */
      free(llink->l$name);      /* before ptr freed */
      free(llink);
      ++n;                      /* count matching names */
    }

    return n;
}
```

This merely requests a directory, and stores the list of names returned in a linked list, making use of the C language *struct* facilities, together with the C libarary dynamic memory allocation routines. When it has finished collecting the directory listing, it executes the given function once for each file name found. In addition, a 'selector' facility is provided.

The *strdup* routine used by *devglob* is merely:

```
#include <stdio.h>
/*
 * strdup - allocate a copy of string
 *
 * returns as function value the
 *  newly allocated string
 *
 * NULL if enough memory is not available
 */
char *
strdup(string)
char *string;
{
  extern char *malloc();
  char *newstr;

  newstr = malloc(strlen(string)+1);
  if (newstr != NULL)
    strcpy(newstr,string);
  return newstr;
}
```

Checking up on Commodore's wildcards

The trouble with a Commodore drive's interpretation of * is that any suffix following it is ignored.

The ***.c** example given above, for instance, would be interpreted by a Commodore drive as matching all files on the drive, not just those ending in **.c**.

To provide a wildcard facility of the sort we would like then, we need to check up on the results of the Commodore disk drive wildcard matching.

Because the C language and the Power C implementation both support *recursion*, a nice wildcard matching routine is not all that difficult to write.

Recursion refers to the ability of routines in a computer language to invoke themselves. To implement this, the language must create an entirely new environment (i.e. set of local variables) for each invocation of a routine. This means it does not matter when a routine calls itself, either directly or indirectly.

The recursion is needed as an easy way to check that all possible ways of expanding any occurrences of * are considered before the pattern is rejected.

For example, the pattern **a*x?o** matches the string **axyoxyo**, but it is necessary to reject the first matched x?o, and go back and look for another.

Our wildmatch routine, to support * and ?, is:

```
/*
 * wildmatch - do moderately primitive
 *   wild card match
 *
 * returns non-zero as the function value
 *   if given pattern matches string
 *
 * pattern can contain the following
 *   special characters:
 *
 *   * - match any number of any characters
 *   ? - match exactly one of any character
 *
 *   all other characters are matched
 *   literally.  '*' and '?' cannot be
 *   matched literally.  Pattern must match
 *   entire string.
 */
#define NO 0
#define YES 1

wildmatch(p,s)
char *p;  /* pattern, as a literal string */
char *s;  /* string to test as matching entire pattern */
{
  char pc;

  for ( ; ; ) {
    /* until we return by failure or success */
    /* switch on next char in pattern */
    switch (pc = *p++)
    {
    case '*':   /* match any number of any char */
      do {
        /* check all possible suffices */
        if (wildmatch(p, s))
          return YES;  /* at least 1 suffix works */
      } while (*s++);
      return NO;  /* all suffices inconsistent */

    case '?':
```

```
      if (*s == '\0') /* match any real char */
        return NO;
      /* else check next char in p and s */
      break;

    case '\0':  /* pattern ended */
      return *s == '\0';  /* YES only if s ends too */
          /* returning YES unconditionally here would */
          /* check if initial portion of s matched p */

    default:   /* literal match of char in pattern */
      /* ASSERT: if *s == '\0' we return (NO) */
      if (*s != pc)
        return NO;

      /* else check next char in p and s */
    }
    ++s;  /* next char in s */
  }
}
```

Because each invocation of *wildmatch* has its own private copies of **p** and **s**, and doesn't disturb the values known to the one calling it, this provides an automatic method of keeping track of the success of each possible suffix to the string thus far matched.

With some care, a non-recursive solution to matching this simple definition of patterns could probably be written, but the algorithm given here is easily extendable to more complex wildcard features. These include characters classes, typically represented by enclosing a list of characters in square brackets ([and]). For example:

```
*.[ch]
```

would match file names ending with either **.c** or **.h** extensions. Other extensions to pattern matching include allowing the repetition of any element, not just the 'any number of any character' as implied by *.

An excellent discussion of pattern matching techniques is included in the classic programming text *Software Tools*, by Brian Kernighan and P.J. Plauger [1], and a sample implementation of the algorithm is actually included on the Power C distribution disk as the program *find.c*.

Putting it all together...

We now combine *devglob* with a call to *wildmatch*, and produce the routine *glob*

```
/*
 * glob.c - expand wildcard filenames
 */
#include <stdio.h>
/*
 * select - double check names matched
 *   by 1541(etc.) dos
 *
 * (used by glob function)
 */
char *savepat = NULL;
static select(name)
char *name;
{
  return wildmatch(savepat, name);
}

/*
 * glob - execute given function for each
 *   file on the work device matching
 *   the given pattern
 */
glob(pat,func)  /* return quit/continue status */
char *pat;       /* filename pattern to expand */
int (*func)();   /* function to call on matches */
{
  int select();
  extern int sprintf();
  extern char *malloc();
  unsigned inbasic; /* flag if linked as basic program */
  unsigned dev;
  char *p, *endp;

#define wrkdev (*(char *)0x17fc)
                  /* device set by shell work command */
#define krndev (*(char *)0xba)
                  /* device last used for kernal OPEN */
  dev = wrkdev;
  /* ASSERT: sprintf needed by devglob */
  inbasic = &sprintf > 0x880;
  if (inbasic) dev = krndev;

  if (pat[0] == ':')  /* names won't contain drive */
    p = pat+1;
  else if (pat[1] == ':')
    p = pat+2;
  else
    p = pat;
  /* make pat accessible to select */
  savepat = strdup(p);

  /* remove =typ from end of pattern */
  /* since it won't appear either */
  p=savepat+strlen(savepat);
  endp=p-4;
  while (p > endp && p > savepat)
    if (*--p == '=') {
      *p = '\0';
      break;
    }
  return devglob(dev, pat, func, &select);
}
```

And taking it on the road

A canonical use of the file name matching *glob* routine is to produce a selective list of matching names. This is similar to the **ls** command of the UNIX system. **ls** is already implemented slightly differently in the Power C shell, so we have named our version **lf**.

It's quite simple really. It just calls our *glob*, using as the function for each file name a routine that simply echoes its argument.

```
/*
 * lf - list file names
 *   selected by a list of patterns
 */
#include <stdio.h>
/*
```

```
 *  function to call for each name
 *   matched by glob
 */
int
echoname(name)
char *name;
{
  printf("%s\n", name);
}

main(argc,argv)
unsigned argc;
char **argv;
{
  extern int echoname();
  unsigned i;

  if (argc < 2)           /* if nothing specified */
    glob("*",&echoname);  /* provide a default action */
  else
    for (i = 1; i < argc; ++i) /* process each pattern */
      glob(argv[i],&echoname);
}
```

A slightly more interesting thing to do is augment a program that takes a list of file name arguments so that it can use patterns instead. This can save much guesswork and/or typing, as in the following program which searches for and prints occurrences of a string in a list of files (patterns).

```
/*
 * findstr.c - string search program
 *
 *  This program will print all occurrences
 * of the given string in the specified files.
 *
 *  The files can be specified as a list of
 * patterns.
 */
#include <stdio.h>

#define MAXBUF 250

char *index();
static char *matchstr = NULL;

main(argc, argv)
unsigned argc;
char *argv[];
{
  int findstr();
  unsigned i;

  if(argc < 3){
    fprintf(stdout, "%s: <match-string> file(s)\n",
      argv[0]);
    exit();
  }
  matchstr = argv[1];
  for(i=2; i<argc; ++i)
    glob(argv[i],&findstr);
}

static char buf[MAXBUF] = {0};

findstr(filename)
char *filename;
{
  FILE infid;
  char *p;
  char c1;
  unsigned length, lineno;

  c1 = *matchstr;
  length = strlen(matchstr);
  infid = fopen(filename, "r");
  if(infid == NULL) {
    fprintf(stdout, "cannot open %s.\n", filename);
    return;
  }
  for (lineno=1; fgets(buf, MAXBUF, infid); ++lineno) {
    for (p=buf; (p=index(p, c1)) != NULL; ++p) {
      /* consider all occurrences of c1 */
      if(strncmp(p, matchstr, length) == 0) {
        printf("%s %d:%s", filename, lineno, buf);
        break;  /* stop after first match on line */
      }
    }
  }
  fclose(infid);
}
```

The same modification could be made to the programs which come on the Power C disk, such as *print.c*, *format.c* and *find.c*.

Improving performance

A noticeable performance improvement can be made by replacing the C *gdirline* routine with one written in assembler. This is primarily because the C library *fgetc* routine does a relatively expensive CHKIN Kernal call for each character.

```
C/ASSM - version 2.0 - 10/24/85
gdirline.a:
; gdirline - C callable function reads a line from directory "load" format
                ; int gdirline(buf,fid)
                ; char *buf               ;
                ; FILE fid                ;
                ;
; returns as function value "basic line number" ("number of blocks" in
; directory line)
;
        FFC6    chkin       =     $ffc6
        FFCC    clrchn      =     $ffcc
;
        033C    argstk=$033c
        004C    savex=$4c
        00FB    bufp=$fb                  ; and $fc
        004D    fid=$4d                   ; for chkin
        004E    savey=$4e
;
                            .ref  c$funct_init
                            .ref  c$getchar
                            .def  gdirline
 ;
 ; first, call c$funct_init for quick setup
 ; copy passed parameters from transfer area (cassette buffer) to zp
 ;
 0000  20 00 00 gdirline    jsr   c$funct_init
 0003  86 4C                stx   savex
 0005  BD 3C 03             lda   argstk,x
 0008  85 FB                sta   bufp        ; output buf
 000A  BD 3D 03             lda   argstk+1,x
 000D  85 FC                sta   bufp+1
 000F  BD 3E 03             lda   argstk+2,x  ; low byte
 0012  85 4D                sta   fid         ; of fid arg
 0014  AA                   tax              ; set logical input device
 0015  20 C6 FF             jsr   chkin
```

```
 0018  20 42 00           jsr   cget ; skip "link"
 001B  20 42 00           jsr   cget
;
; now get "basic line number" and save as normal return value
;
 001E  20 42 00           jsr   cget
 0021  A6 4C              ldx   savex
 0023  9D 3C 03           sta   argstk,x   ; low byte
 0026  20 42 00           jsr   cget
 0029  A6 4C              ldx   savex
 002B  9D 3D 03           sta   argstk+1,x  ; and high
;
; read rest of line into buffer, until a zero byte or bad status encountered
;
 002E  A0 00              ldy   #0
 0030  84 4E              sty   savey
;
 0032  20 42 00 loop      jsr   cget       ; next char
 0035  A4 4E              ldy   savey
 0037  91 FB              sta   (bufp),y   ; save it
 0039  E6 4E              inc   savey
 003B  C9 00              cmp   #0         ; zero?
 003D  D0 F3              bne   loop       ; no
 003F  4C CC FF           jmp   clrchn     ; yes return
;
; cget - call the C library internal routine c$getchar to read a character from
; the current chkin input file. c$getchar must still be passed the logical file
; number in the 'a' register so it can check whether EOF has been reached on
; that file yet.
;
; c$getchar returns with carry set if EOF reached, otherwise it
; returns the next character from the file in the 'a' register. cget is local
; to this gdirline function, so it arranges to return from gdirline with a
; function value of EOF when end-of-file is reached.
; Normally, cget returns the
; next character from the current file in the 'a' register.
;
 0042  A5 4D    cget      lda   fid
 0044  20 00 00           jsr   c$getchar
 0047  B0 01              bcs   eof
 0049  60                 rts
;
 004A  68       eof       pla ; return EOF from gdirline
 004B  68                 pla ; pop return address
 004C  A6 4C              ldx   savex    ; store EOF (-1) as return value
 004E  A9 FF              lda   #$ff
 0050  9D 3C 03           sta   argstk,x
 0053  9D 3D 03           sta   argstk+1,x
               ;
               ; and return from gdirline
               ;
 0056  4C CC FF           jmp   clrchn

 End of assembly, 0 errors
```

The *gdirline.a* file can be assembled using the public domain *C/Assm* assembler to produce a *gdirline.o* file which can be linked in place of the compiled version of *gdirline.c* to make *glob* run about twice as fast.

A bug (oh no!)

A word of caution: Commodore drives do not work well with an output file open at the same time as the directory. Sometimes (but not always) a program using the *glob* function, and creating an output file on the same device will have its directory reading terminated prematurely with strange errors. The same phenomenon can be observed when redirecting the output of the Power C built-in **l** command, as in: `l >directory.tmp`

Very often, but not always, the last few files on the disk will be absent from the listing created in the output file.

Exercises for the reader

Well, it wouldn't be fair to finish this discussion without leaving the reader with some food for thought, would it? So I'll end with a few suggestions as to how you may want to alter *glob* for your own applications.

Very often when specifying a file name to a Commodore disk drive, it is necessary to suffix it with the file type. Under certain circumstances, then, this **,s**, **,p**, or **,u** (or even **,r** or **,c**) suffix gives extra information about the file name. It is similar to the file name extensions of MS-DOS, although usually it is optional. Still, it might be nice to have it on the end of the file name passed to the function called through **glob**.

Perhaps, then, if the pattern passed to *devglob* includes a suffix like **,p**, or **,***, or **,?** attention should be paid to the file type and it should be put on the end of the file name passed on to the function.

Similar treatment could be afforded **?:**, ***:**, or **0:** or **1:** to specify a directory. A way could even be worked out to treat the device number as part of the pattern.

On UNIX, the list of file names matching a pattern is always sorted alphabetically. Thus could be done in *devglob* as well.

glob is presented here as a routine. This *glob* routine could be used to create a *glob* program that could, in a limited fashion, 'glob' other programs or commands.

```
glob <command> [<pattern>...]
```

could expand the patterns into the lists of matching file names, and cause:

```
<command> [<filename>...]
```

to be executed. This can be done by writing the expanded command line to the screen, and stuffing the keyboard input buffer to cause it to be read by the Power C shell after the *glob* program exits. Of course, if the line would end up being greater than eighty characters in length, it would need to be truncated.

As presented here, the *glob* facility can be useful to save typing, and allow searching for files. As hinted at earlier, the pattern-matching principle need not only be used for file names, but can be used when matching other types of strings.

[1]Kernighan, Brian W. and Plauger, P.J., *Software Tools*, Addison-Wesley, 1976.

Two Assemblers For GEOS

A comparison of GeoCOPE and Geoprogrammer

by Francis G. Kostella

GeoCOPE is available for $20 (US) from:

Bill Sharp Computing
P.O. Box 7533
Waco, TX 76714

Geoprogrammer is available from:

Berkeley Softworks
2150 Shattuck Ave.
Berkeley, CA 94704

I'd like to present you with a brief overview of two assemblers that run under GEOS, and then throw a short argument at you as to why *you* should be writing programs for GEOS.

If you've written programs for GEOS with a non-GEOS assembler, you are probably familiar with the time-consuming hassle that the programmer must endure in order to convert his object file into a runnable GEOS file. Besides the object file needing to be manipulated in order to work with GEOS, one must constantly exit GEOS to make modifications and wait while the assembler creates the new file, usually without the speed that the TurboDOS adds, then re-boot the GEOS system.

What the GEOS programmer needs is a system that operates while in the GEOS environment and outputs GEOS-ready files. There are presently two packages (that I know of!) that do this: **geoCOPE** from Bill Sharp Computing and **Geoprogrammer** from Berkeley Softworks.

Looking into GeoCOPE

GeoCOPE is the less complex of the two and was the first GEOS-specific assembler released. The **COPE** system has two main programs, *Editor* and *copeASM* (the assembler).

The nicest part of the **COPE** system is the *Editor*. **COPE** uses its own unique structure for its source files, and each page of a source file can hold up to 8K of text. A feature that I found very useful, was the *Editor*'s ability to make the source files either GEOS SEQ or VLIR. Thus, my system equates file would be SEQ structure and would be **.included** in the main VLIR source file. A VLIR file can have 127 records, and at 8K per record, you can see that each of your source files can get very large if needed.

A few other features of the *Editor* that I liked and found very useful were the ability to set a Bookmark, so that you could return to the last line edited, and an Autosave feature that automatically updated changes when selected. The biggest advantage of the *Editor* is that it is fast! Unlike *geoWrite,* you can quickly scroll up or down through a document, and since it doesn't support fonts or font styles you're not left waiting while the system calculates 24-point bold outlines. The *Editor* also supports Text Scraps, has a Save & Replace function, and allows you to save single pages of a VLIR source file as single SEQ source files, and SEQ as pages of VLIR files. The **COPE** *Editor* also allows you access to Desk Accessories (the system comes with one: *HexCalc* a hex/dec/bin calculator) and works with an REU. The only thing missing from the *Editor* is the ability to use tabs to offset the opcodes and comments from the left margin.

One thing to be aware of when using **COPE**'s *Editor*, is that it stores each line of the source file as a null-terminated string. If you use the Text Scrap function to move text to or from *geoWrite*, you'll run into a few problems. Pasting a Scrap into a **COPE** file from a *geoWrite* document is simply taken care of by entering CRs where the line ends should be. But *geoWrite* will choke on **COPE** Scraps (the *Text Manager* has no trouble with them, though). I've managed to get around this problem by writing a simple filter program that strips out all the nulls except the final one.

COPE supports 21 different pseudo-ops, from the typical **.BYTE** and **.WORD** to some GEOS-specific ones like **.ICON** for defining header icons and **.SEGMENT** for mapping out any VLIR modules. **COPE** also allows you to define macros with the **.MAC** and **.MND** directives.

Labels can be up to 32 characters in length and are case sensitive. **COPE** also allows you to use local branch labels - up to 32 outstanding (unresolved) local labels are permitted. All the usual arithmetic and logical operators are supported (*, /, +, -, AND, EOR, OR).

The major advantage that the COPE system has is its simple and direct approach. Within an hour of reading the manual, I had an application up and running. The manual itself only details the features of the *Editor* and *copeASM*. The examples are few, but cover all the specifics. If you've used MADS, you'll adapt very quickly as the two are very similar.

The manual together with the sample files will have you writing VLIR applications in no time at all. By using the **.SEGMENT** directive, you simply indicate the end of one VLIR module and the start of another. The nice thing about this approach is that labels in all the different modules are global and the need for jump tables or duplicate label definitions is eliminated.

COPE files are easy to maintain and update. I've kept all my COPE source in VLIR structure: the first page holds the header block definition and an index to the entire file, the second page holds my constants and equates, the third page holds all of my macros, and the source code begins on page four. Writing even the largest of applications, I've never gone over 20 VLIR pages of source.

Once you have your source code together, you load the assembler and select the file. *CopeASM* is a two-pass assembler that will assemble to disk and print any errors to the screen. You may list the assembly on its second pass and can turn this listing on or off. *CopeASM* also allows you to pause the listing if needed. If the file assembles without any errors, you can exit to DeskTop and run the file. If you do have errors, you'll have to pause the assembler and scribble them down. This is the weakest part of the system - an option to output to an error file is sorely needed. Furthermore, attempting to pause the screen listing will sometimes scroll the errors off the screen, forcing you to reassemble just to see the errors.

If you're just getting your feet wet with GEOS, GeoCOPE is the perfect place to start...

Presently, COPE can only handle up to 5000 characters in its label table, so you'll have to keep those labels short and use plenty of local labels if you are assembling a large application. Another limit of *copeASM* is that it can only assemble 8K sections of code at a time. This isn't as big a problem as it may seem, though: you can simply divide the file up into VLIR modules and load them in as part of your initialization routine. My favorite approach here is to put all of my tables, graphics, and fonts into one or more VLIR modules and load them in right after drawing the intro screen.

Although I haven't done any rigorous comparisons of assembly times, I've got one good example: the original version of CIRCE was assembled with the MADS assembler, then converted to GEOS format. The amount of time taken between loading the MADS assembler and loading the assembled and converted file from the DeskTop was slightly under 35 minutes. After converting the source files to COPE format, the time between loading the COPE assembler and loading the assembled application was just about four minutes! Now I've got to admit that this was on an REU, but even without it, the assembly took about six or seven minutes. (Besides, MADS didn't support the REU.)

If you're just getting your feet wet with GEOS, COPE is the perfect place to start. The system easily handles smaller programs and is fairly fast and easy to maintain. But once your applications begin to grow in size, that 5,000 character symbol table begins to fill up fairly quickly. Once you get to this point, you should consider **Geoprogrammer**.

The first thing that strikes you when you open the **Geoprogrammer** package is the 450-page manual. If you have a number of GEOS programs, you are perhaps used to the sometimes simplistic documentation that presents you with the "this is a disk, put the disk in the drive..." level of information. I was very pleasantly surprised to open this manual and read through it without ever having my intelligence insulted. The first two dozen pages do contain the basic info for first time GEOS users, and there is a chapter devoted to an overview of the **Geoprogrammer** system, but the rest of the manual is filled with loads of useful information. Granted, the info is not all organized as well as it might be ("Now, *where* is that part about bitwise exclusive-or?!") but generally you'll be able to find what you need with a little persistence. In addition, the appendices of the manual contain detailed listings of all the system constants and variables, along with a hardcopy listing of the macro and sample source files included on disk.

The **Geoprogrammer** disk itself is a flippy that includes, besides the sample files and system symbols and macros, the three programs that make up the **Geoprogrammer** system: *GEOASSEMBLER*, *GEOLINKER* and *GEODEBUGGER*. No, there is no editor here. Unless you have a text editor that handles *geoWrite* files *[such as Q&D Edit, written by Kostella and Buckley and available from RUN - Ed.]*, you'll have to edit your source files with *geoWrite*! And **Geoprogrammer** is a two-drive system; you *can* use it with one drive, but the amount of disk swapping involved will quickly convince you that that second drive is worth the money. Better yet, an REU is not only a fast second drive that makes using *geoWrite* tolerable, but it will allow you to use *GEODEBUGGER* to its fullest.

That being said, let me give a brief overview of the assembly process. Once you've edited all of your source, you assemble each of the source files into *.rel* relocatable object files. Then you load a linker file (also a *geoWrite* document) into *GEOLINKER* to link together your *.rel* files into a runnable GEOS file on disk. Now you can load *GEODEBUGGER* to test and debug your program. Sounds simple, eh? Well, there's a lot more going on here than would appear. **Geoprogrammer** gives you access to some powerful features and abilities that you may find that you won't want to do without once you've gotten used to them.

First off, *GEOASSEMBLER* is a two-pass assembler that supports a number of useful features: conditional assembly, macros, local labels, the ability to parse complex algebraic expressions, and the ability to pass symbols to the linker or debugger. *GEOASSEMBLER* will also output an error file to disk (in *geoWrite* format, of course!), if needed, for each file assembled.

When *GEOASSEMBLER* starts assembling a file, it uses three counters to keep track of the code: **.zsect** for zero page ram, **.psect** for program code, and **.ramsect** for uninitialized data. If you're lazy like I am, when you need a new variable, you just add it somewhere in the current section of code instead of adding it to a separate section of code for variables. By using the **.psect** and **.ramsect** directives, you can add variables just about anywhere like this:

```
.ramsect
MyVariable:    .block 1
.psect\b
```

When the assembler encounters this construction, it will give the label *MyVariable* the address of the current address of **.ramsect** (which can be set by the **.ramsect** directive or in the linker file). The **.ramsect** section defaults to the RAM following the last byte of code, thus we don't end up assembling uninitialized variables and add to the length of the program. Perhaps not a big deal, but when you have a few hundred bytes of variables it becomes noticeable.

Another useful feature of the assembler is the 16-bit expression evaluator (*GEOLINKER* also uses the same evaluator). Besides the usual arithmetic, the evaluator handles a number of logical operators: the manual lists thirty of them. I usually keep away from creating expressions too complex to be understood at one glance, but *GEOASSEMBLER* will let you create some truly bizarre and outlandish expressions if you so desire! But the real power I find here is that you can easily create data tables with a few easily changed constants at the root of some complex expressions. Perhaps this doesn't seem that unusual, but I've been able to create expressions that all the other assemblers I own have choked on, and I don't miss having to do the math by hand.

You run *GEOASSEMBLER* from DeskTop and select the file to assemble from the typical 15-file dialog box. Once you've selected the file to assemble, you are given a choice of drive for the output file, then the file is assembled. The output file is the same name as the source file but with a *.rel* appended. When this is done, you can quit to DeskTop, assemble another file, or open the error file (i.e. enter *geoWrite*) if one was generated. A friend of mine who beta-tested the version 2.0 package tells me that *GEOASSEMBLER* V2.0 will allow you to go directly to the linker, and that it is not limited to selecting only the first 15 source files on disk.

Once you've assembled all the *.rel* files in your program, it's time to use *GEOLINKER*. *GEOLINKER* does more than just connect separate object files together. First of all, *GEOLINKER* uses a link file to determine the structure of the output program, be it GEOS SEQ, VLIR, or CBM, which is a 'regular' object file. Secondly, the linker will add the header to the file. The linker will also cross-resolve all label references between the different *.rel* files; if a label is defined in two different files, but is not referenced in a third file *GEOLINKER* will not flag an error.

One thing I would like to mention here is that although the assembler and linker accept symbol names up to 20 characters in length, only the first eight are significant. This is not really a problem, but you should remember that there are a few hundred system symbols in the *geosSym* file usually *.include*d during assembly.

I once wrote a routine named *DeleteRegion* that was only called from a routine in another source file. *DeleteRegion* never seemed to be called, but the disk would go active for a second when it should have been called. The debugger only shows the eight significant characters of the label when you list the code, so I couldn't imagine what was happening. But the debugger also lets you view the label name by its hex address, and upon examination, this label appeared somewhere in the Kernal jump table. The routine that was being called was the Kernal routine *DeleteRecord*! Luckily I didn't have any VLIR files open....

GEOLINKER will also allow you to output a separate symbol table (again, a *geoWrite* file) to the drive of your choice. Like *GEOASSEMBLER*, if there are any errors, you have the option of directly opening the error file after linking. One thing to note here is that when there are more than 99 errors, the system will sometimes have a fatal crash.

When *GEOLINKER* creates the file on disk, it also writes a special *.dbg* file to disk for use by the debugger - more on this later.

The manual does not include specifications for either *GEOASSEMBLER* or *GEOLINKER*, otherwise I'd be happy to include a list here. If you do run into problems assembling and linking very large files, you can always use the **.noglbl** and **.noeqin** directives to cut down the number of symbols passed to the linker. One way to quickly cut down on the number of symbols is to use **.noglbl** and **.noeqin** before **.include *geoSym*** (the system labels and equates) and **.glbl** and **.eqin** after. Of course you don't get anything for free, and these symbols don't get passed to the debugger!

A sneaky trick I use when doing this, to get the system symbols to the debugger when writing a large VLIR application, is to assemble the *geosRoutines* and *geosMemoryMap* and link them into one of my modules that isn't using very many symbols. Each module has its own set of symbols and is selected in the debugger by the **setmod** command. If you're debugging a stretch of code that uses a lot of system calls, just reset the module priority to the module with the system symbols.

Alternately, I find that just defining the pseudo-registers as global equates helps a great deal when debugging.

Another potential problem I've noticed that is not documented, is that when linking files from two drives, the linker seems to search for the file on the current drive first, then checks the other drive. The problem here is that if you have two different versions of one particular *.rel* file, let's call it *beta.rel*, and this file is supposed to be linked after a file called *alpha.rel* that is not on the current disk, when the linker switches drives to find *alpha.rel*, it does not switch back to the original drive, potentially linking the wrong *beta.rel*. I don't have any empirical evidence for this, but if your modifications don't seem to be appearing in your new version of a program, try moving all of the *.rel* files in one VLIR module to the same disk.

Once you've assembled a GEOS program, it's time to load *GEODEBUGGER* - here's where the fun begins! *GEODEBUGGER* is a sort of 'shell' that uses NMIs to take control of the machine and allows you to debug the program (or examine the Kernal) in an almost interactive environment. Load your program from the debugger. Need to tweak some values or check why that branch isn't being taken? Just bang on the RESTORE key and you're in the *GEODEBUGGER* again. *GEODEBUGGER* is the best thing about this package; you can set breakpoints, alter stack or register values, and access the disk drives almost like a sector editor.

Geoprogrammer along with an REU will dramatically increase your output and capabilities...

There are actually two versions of GEODEBUGGER. When you load the program, it first checks for a REU. If there is one connected, the full debugger is loaded into the REU, otherwise the mini-debugger is loaded into RAM from $3E00 to $5FFF. Needless to say, the REU super-debugger is the preferred option.

What makes the debugger truly useful is the ability for it to use the *.dbg* files generated by the linker. These files contain a list of symbols and their addresses. This way, while in the debugger, you can list and modify a section of code using labels from your source files. Of course, your changes are not saved, but this allows you to try out different things without constantly reassembling the program.

GEODEBUGGER is basically a machine language monitor with plenty of features. One of the most powerful of these in the super-debugger is the ability to define macros. Most of the commands in the debugger are actually system macros composed of a number of macro primitives. *GEODEBUGGER* allows you to define up to 1,000 bytes of user macros. These user macros can be made up of the macro primitives or system macros. A macro file with the same name as your application will be automatically loaded along with your application. Optionally, you can define a default set of macros and an autoexec macro to run when the debugger is loaded. For example, the linker always passes a few of the system variables to the debugger, but I have no use for them, so my autoexec macro removes these from the symbol table, like this:

```
.macro autoexec      ; name
   clrsym Pass1[cr] ; eliminate symbol
   clrsym picW[cr]
   clrsym picH[cr]
.endm
```

There are dozens of other commands, but there are problems with some of the memory commands, most notably FILL, which doesn't work at all. My beta-test friend tells me that this bug has been eliminated in the 2.0 version. The only other thing that one could desire would be a more detailed description of the macro primitives in the manual.

Overall, the **Geoprogrammer** package is nicely put together, and along with an REU will dramatically increase your output and capabilities. Most especially, the debugger will teach you about programming for GEOS by allowing you to examine any GEOS program and the GEOS Kernal in detail.

If you're new to assembly language, I suggest that you give writing GEOS programs a try. A common problem for beginners is the need to develop a set of routines to perform common functions; i.e., printing text and graphics to the screen, moving large chunks of memory around, disk access, and string input to name just a few. When you're just starting out, you basically begin with nothing, and until you've accumulated enough experience to write code to perform some of the above functions, your ability to write useful programs is hindered. When you code for GEOS, all of these basic functions are always available. You can concentrate on writing the 'heart' of the program without getting bogged down in minor details. By getting programs up and running quickly, the beginner will (hopefully) form positive associations with assembly language, instead of thinking of it as some arcane art which is painfully learned!

If you're interested in writing GEOS programs, you must get an assembler package that runs in GEOS. If you're not sure how far you want to go I suggest you get the **geoCOPE** assembler from Bill Sharp Computing. The price is good and the system direct and uncomplicated. If you later decide that you need more power and you have a two-drive system (or REU!) and can afford the price, go for **Geoprogrammer**. If you're an experienced programmer, I suggest that you go straight to **Geoprogrammer** or get them both.

Berkeley Softworks should be commended for releasing such a nice package, especially considering the relatively small market for products of this nature. Unfortunately, they don't seem as if they're going to release version 2.0 any time soon. One only hopes that they would at least consider doing a mail-in upgrade for present users.

NewsBRK

Format Executive Version 4.0: Powersoft has announced the release of **Format Executive Version 4.0.** Format Executive is the first and only comprehensive disk format and file transfer program for the Commodore 128 and 128D. Format Executive now allows a Commodore 128 or 128D computer with 1571 or 1581 drive to read, write and format over 150 different 3.5" or 5.25" MS-DOS (PC-DOS), CP/M-80, CP/M-86, Commodore CP/M and Commodore DOS (PRG, SEQ, USR, REL) disk formats. This means the Commodore 128 can now transfer files back and forth from almost any CP/M or MS-DOS microcomputer. The manual also details how to use Format Executive Version 4.0 to transfer files from machines such as the Commodore Amiga, the Atari ST and the Apple Macintosh.

Format Executive Version 4.0 features include: high speed burst file transfer technique, file transfers between all formats, Commodore PETASCII to true ASCII conversion, linefeed adjustment, wildcard support (?,*), single drive, multiple drives, dual drives, RAMdisk, and hard drive support; CP/M user area support, 1581 partition support, and automatic disk login. Backup disks are permitted. All Commodore drive devices are supported: 154x, 157x, 158x drives, and 17xx RAMdisks. Format Executive Version 4.0 will permit transfers of any ASCII or OBJECT file of any length at burst speed.

Format Executive is compatible with C128 system enhancements such as **JiffyDOS**. A complete list of supported formats is available on QuantumLink in the file: FE-FMTS. Format Executive Version 4.0 is available for the Commodore 128 or 128D with 1571 or 1581 drive. Send check (2 wks.) or money order for $59.95 plus $3.50 S&H (COD add $3) to: Powersoft, Inc., P.O. Box 7333, Bradenton, FL, 34210. On QuantumLink: POWERSOFT

CP/M Productivity Software: The Public Domain Software Copying Company is offering CP/M users an exclusive repackaging of **WordStar** version 2.26. PDSC says that CP/M users are often left out of current software releases. PDSC says that it is committed to providing CP/M users with the most productive CP/M software available.

WordStar version 2.26 can turn a Commodore 128 into an effective, powerful word processor including features such as MailMerge, which allows users to merge text and/or data files to generate form letters, boilerplate text (text created from files of pre-existing, commonly used sections of text), mailing lists, and large documents.

PDSC is offering this as the first of many classic software programs formatted for the Commodore 128. This Commodore 128 - CP/M disk edition comes complete with an Osborne 1 User's Reference Guide that explains how to use WordStar, and also has sections on the CP/M operating system.

WordStar version 2.26 is IBM data-compatible using readily available conversion software including **Big Blue Reader** and **Uniform**. WordStar version 2.26 and manual are available for $39.95 plus $4.50 for postage and handling.

The User's Guide also describes how to use other programs available from PDSC: **SuperCalc** and **MicroSoft BASIC,** which are $20 each when ordered with WordStar version 2.26 ($39.95 each plus $4.50 for postage and handling if ordered separately).

Also included in the WordStar version 2.26 package is a set of Key Fronts. More useful than a quick reference guide, these self-adhesive letters attach to your keyboard and include all the commands you need to use the special word processing functions of WordStar. For more information, call or write: PDSC, 33 Gold St., Suite L3, New York, NY, 10038, (800) 221-7372, (212) 732-2258.

FORTRAN Compiler for C64: Thirty years ago, FORTRAN was the first high-level language. Today it remains one of the most universally-used programming languages. **Fortran 64** supports more than 45 statements and functions and is a practical, economical and convenient way for users to learn Fortran on the C64. Fortran 64 is aimed at the student and novice programmer who wants to use this mathematically-based language.

Fortran 64 includes a built-in editor, compiler and linker, and creates a fast standalone program. Once completed, the program module may be run without Fortran. Subroutines and functions may be compiled separately from the main program. Input and output may be free-form or formatted, and the user has access to the 6502 registers, Kernal and machine language routines. Fortran 64 carries a suggested retail price of $39.95. Other language products from Abacus: Ada, Assembler, BASIC compilers, COBOL, C and Pascal. Contact: Abacus Software, 5370 52nd St. SE, Grand Rapids, MI, 49508, (616) 698-0330.

SYSLAW: A Legal Guide for SysOps: A new book from LLM Press explains the legal rights and responsibilities of sysops, the people who operate computerized bulletin board systems (BBS). Written in clear English by Jonathan Wallace and Rees Morrison, two lawyers who are both veteran sysops, the one-hundred page book covers all aspects of the emerging area of SYSLAW.

"More than 4,500 computer bulletin board systems are running in this country, but the legal rights and risks of their sysops have never been explained", says Jonathan Wallace. "Hundreds of unknowing sysops set up BBS's on their home computers each year," according to co-author Rees Morrison, "and they don't know what can happen to them legally, or what they can do."

SYSLAW, or sysop law, for those who run bulletin board systems, concerns the legal consequences for those who run BBS's. The total number of people who dial the thousands of bulletin board systems is in the several hundred thousands, and the legal issues are plentiful. More important, the law affecting sysops is not merely unsettled, it is unestablished.

Significant legal issues have arisen in connection with BBS's, but neither courts nor legislators have come to grips with this technology. Scattered state and federal statutes can affect a BBS, and the common law development of the area has been sparse and confused. Given the dynamic activity of BBS's, the time was ripe for a careful treatment of the many potential problems, which include:

- What if someone posts copyrighted material on the board?
- Does it matter if you charge users or accept ads?
- What if you delete a crucial message by accident?
- Can you bar someone from using your BBS?

Being unsettled, SYSLAW might deter the development of this telecommunications explosion. The authors believe that BBS's offer an exciting, progressive and influential medium, even in its infancy, that will be promoted by clarity of law and an understanding of sysop's rights and responsibilities.

The softcover manual costs $19.00 plus $2.00 for postage and handling. Orders and cheques may be sent to LLM Press, 150 Broadway, Suite 610, New York, NY, 10038.

SimCity, The City Simulator: Maxis Software announces the release of **SimCity, The City Simulator**, for C64/C128. When you enter SimCity you take on the role of Mayor and City Planner of a sophisticated simulated city. You zone land, balance budgets, install utilities, manipulate economic markets, control crime, traffic and pollution and overcome natural disasters. You control the fate of the city.

Design, plan and grow your own utopian dream city from the ground up, or take over any of eight included pre-built cities on the verge of disaster. Scenarios include: San Francisco, CA 1906, just before the great quake; Tokyo, Japan 1957, just before a monster attack; and Boston, MA 2010, just before a nuclear meltdown. Watch the disaster occur, gather your funds and information and bring the city back to life.

The city is alive: you see traffic on the roads, trains on the rails, planes in the air, even football games in the stadium. You see population levels rise and fall, residential areas develop from single family homes to condos to slums. Watch commercial and industrial areas grow or decline depending on your skill as a strategic city planner.

While *you* do the planning and zoning, it is the Simulated Citizens, a.k.a. Sims, who move in and actually build the city. Sims live, work, play, move, drive, and complain about taxes, traffic, taxes, crime and taxes - just like humans.

SimCity is primarily a constructive game, but those who can't enjoy a game without destruction can wipe out a city through terrorism, financial mismanagement, or by evoking a natural disaster such as an earthquake or monster attack.

SimCity comes with extensive documentation including a User Reference, an explanation of the inner workings of the simulation, and an essay on the History of Cities and City Planning.

SimCity is distributed by Brøderbund and carries a retail price of $29.95. Maxis Software specializes in System Simulations, a new type of entertainment software, with emphasis on quality graphics and sophisticated simulation techniques. Maxis Software, 953 Mountain View Drive, Suite #113, Lafayette, CA, 94549.

C128 CP/M Software from Cranberry Software Tools: Cranberry Software Tools offers a variety of products including a public domain diskette containing what the vendor describes as "an entirely new CP/M environment for the C128". Included on the disk are:

- NEWSYS.COM - generates the 12/6/85 version of CP/M 3.0
- CONF.COM - CP/M 3.0 configuration utility
- CCP.COM - the famous CCP104 upgraded command environment
- HIST.COM - command line editing accessory

The CCP104 environment allows the recall, editing and re-execution of command lines; easy use of user areas; and named reference to user areas. A total of 21 files are supplied, including powerful CP/M file management utilities such as ACOPY, CSWEEP and NSWEEP. Limited documentation is also included. The cost of the disk is just $5.00 postpaid!

Want to try your hand at programming? Cranberry Software Tools' **PD Programming Series** is an inexpensive way to get your feet wet. The following CP/M Language packages cost only $5.00 (!) per disk:

1. Three BASICs: BASIC5, EBASIC, ZBASIC
2. FOCAL - calculator-like stack language (sorry, no documentation)
3. Two FORTHs: UNIFORTH, and FIG-FORTH (sorry, no documentation)
4. LASM - Z80 Assembler
5. SAM76 - unusual symbolic processing language
6. Small-C Interpreter - fun, interactive C environment
7. Small-C Compiler - fast subset of standard C, assembler required
8. Parasol System - powerful and complete, like COBOL
9. Draco System - powerful and complete, like C
10. E-Prolog - rule-based artificial intelligence
11. Algol/M - scientific language like Algol-60

12. Concurrent Pascal-S - has coprocesses like Modula-2
13. PL/0 - miniature compiler, complete with source
14. RATFOR Translator - FORTRAN never looked this good

Cranberry's **Disk Reporter-128** was created with the C128 owner in mind. DR-128 takes advantage of many of the special features found only in the C128's CP/M+. DR-128 will display the contents of your CP/M diskettes sorted by filename, extension, size or data. The contents of all user areas are shown, not just area zero. Unlike many utilities written for the older CP/M 2.2, DR-128 will display the correct amount of free space remaining on a C128 diskette. Best of all, DR-128 will print handy diskette labels on your printer. Just $8.95.

The **AlphaNote Quick Reference System** from Cranberry is a personal text database program for all computers running the CP/M operating system (a special version customized for the C128 is available). AlphaNote has been designed to serve as an "intelligent memo pad" that can assist you in capturing the large number of useful facts that bombard you during the average day.

AlphaNote can store an unlimited number of free-format text notes, with no length restrictions other than the size of your drive. Each note can be associated with several 'key phrases' that can assist you in finding the note after it is stored, much like the card catalog in a library.

Partial keyword matching and case-independent keyword matching are available to extend the power of your searches through AlphaNote data files. No need to memorize lots of keywords - you can choose to search through the main text of your notes as well. You can create and use as many different 'note-bases' as you wish. Using AlphaNote is a snap, with the program's easy-to-use, single-key menu interface.

AlphaNote can display any note using quick, random-access disk techniques. You can call up a full-screen 'directory' of your stored notes as an alternative to searching for a particular entry. Notes may be quickly added and deleted, to better accommodate the short-term importance of this type of data. The special introductory price of AlphaNote is just $9.95 postpaid.

Also from Cranberry Software Tools, **Alpha Text Tools** ($39.95) is an integrated text processing system perfect for home or small business use. The Tools are divided into six major sections: 1) **AlphaText Formatter**, an embedded command text processor with over 30 commands, giving the user total control over margins, page layout, headers and footers, numbering style, and text justification. AlphaText also features mail merge capability, automatic numbering of paragraphs and lists, table of contents generation, and user customizable printer and video control, including on-screen preview of the formatted text. 2) **AlphaEdit Editor**, for effortless editing of both documents and program text. AlphaEdit features over 30 single-key commands and is optimized for use with AlphaText. 3) **AlphaSpell Spelling Checker**, complete with a 34,000 word dictionary. 4) **AlphaFont Printer Enhancer**, which gives owners of Epson compatible dot-matrix printers the capability to produce near letter quality documents in four distinctive fonts. You can design your own fonts or modify the fonts suppplied with the package. Custom symbols may be easily created. Most European languages are supported and fonts are being created for many Middle and Far Eastern languages. AlphaFont can process both right-flowing and left-flowing languages. 5) **AlphaMenu Interactive Environment**, for easy execution of the Tools with just a single keypress. 6) **Alpha Utilities**, which perform dictionary update, text file conversion, and font creation and modification.

The Alpha Text Tools are furnished with an example-filled User's Guide, two quick reference pages, and a Tutorial for new users.

An 80,000 word AlphaSpell dictionary is also available ($10.95) as is a disk with four additional fonts ($12.95).

AlphaShape ($9.95) will reformat documents that have already been formatted, such as the documentation files that usually accompany shareware. AlphaShape reparses such documents, eliminating control characters and changing pagination, blank line sequences, and margins to the user's taste. It also supports a two-column printing mode for dramatic size reduction of the input text.

The **Alpha C Tools** package ($31.95) is composed of: 1) **Source Lister Utility**, that produces appealing listings of source code on the CRT, the printer, or in a disk file. Listings are paginated; page headers contain the input filename, the system data/time, a user-supplied commentary line, and the file modification date/time. Each source line is displayed with a logical nesting level to highlight the hierarchical structure of the code. A handy table of contents lists all procedures and functions. Reserved words can be printed in uppercase, boldface, or both. An outlining feature permits interactive viewing of the code by logical nest level. 2) **Source Reformatter Utility**, that automatically realigns the indentation of the source code according to program logic. 3) **Cross Reference Utility**, that completely maps all non-reserved symbols and provides the same elegant reporting format as the Source Lister. Each member of the C Tools has a flexible UNIX-style command line interface.

AlphaCPP ($19.95) is a macro preprocessor that provides powerful conditional processing features for normal text files, and for languages lacking conditional constructs (like Turbo Pascal V3). Support for the #define, #ifdef, #ifndef, #include, #else, and #undef keywords.

AlphaDump is a file dumping utility that produces formatted output on either the CRT or printer. The user can select five dump formats: binary, octal, decimal, hexadecimal and ASCII. Only $8.95. All prices in U.S. dollars. Contact: Cranberry Software Tools, P.O. Box 681, Princeton Junction, NJ, 08550-0681.

The Potpourri Disk

Help!

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

Loan Helper

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

Keyboard

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

Filedump

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

Anagrams

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

Life

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

War Balloons

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

Von Googol

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

News

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

Wrd

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

Quiz

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

AHA! Lander

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

Bag the Elves

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

Blackjack

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

File Compare

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

Ghoul Dogs

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

Octagons

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

Backstreets

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

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

The ENTIRE POTPOURRI COLLECTION JUST $17.95 US!!

See Order Card at Center