TRANSACTOR: THE MAGAZINE FOR COMMODORE PROGRAMMERS NOVEMBER 1987: VOLUME 8, ISSUE 3

Transactor

In Depth Review:
Commodore's New
1581 Disk Drive

CP/M Mods for the 1581

Transparent C64
ROM Cartridges

Easy–To–Build User Port
to True RS232 Interface

Independent 80-Column
Screens on the C128

80-Column Hi–Res
Colour on the C128

More C128 MMU Secrets!

Amiga Section:

Butterfield: Quote Mode
on the Amiga? It's there!

Event Maker:
Look Ma, No Hands!

Get 1080 Performance
from your 1902 Monitor

Enhanced ECHO Command

BULLET–PROOFING YOUR COMPUTER
Inexpensive surge protection in minutes!

Canada $4.25
USA $3.50

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

Volume 8
Issue 03
Paid Circulation
15,000

Transactor

Amiga Section

Note: before entering programs, see "Verifizer" on page 4

ABOUT THE COVER: Toronto's CN Tower is the tallest free-standing structure in the world. It took just over 3 years for a crew of 1537 to complete the tower, at a cost of $57 million. It was built by CN Telecommunications to overcome the problems faced by firms wishing to transmit their signals northward past the growing Toronto downtown core. The tip reaches 1815 feet 5 inches (553.33m) into the sky – about 5½ football fields. The "doughnut" shaped portion houses most of the communications equipment. The angled perimeter just above is the outdoor observation deck (height: 1136 feet or 346m). One floor above that is the indoor observation deck. On the same level is "Sparkles", a discotheque with a spectacular view of the Toronto skyline, said to be one of the most handsome profiles in the world. Above that is the "Top of Toronto" restaurant, aptly named, which makes one complete revolution every 72 minutes. The smaller pod at the base of the antenna is known as the space deck. It's the highest observation deck in the world at 1465 feet (447m), but only 11 feet higher than the roof of the Sears Tower in Chicago. All decks combined have a capacity of about 400 people, and about 1.7 million visit each year. Ascending or descending, the elevators travel at about 15 kmh, making them the fastest commercial elevators in the world – descending is equivalent to falling in an open parachute. Tip to base, the tower is within plumb by 1.1 inch. On windy days the tip can sway 3 feet in either direction (limited by two 10 ton swinging counterweights mounted on the antenna), and was designed to withstand winds up to 260 mph. All windows have an outer pane, 3/8 inch thick, an inner pane, 1/4 inch thick, are armour-plated, and can withstand pressures in excess of 120 psi. All construction materials were chosen for their fire-proof or fire-resistant qualities. Smoking is allowed in the restaurants only. In the unlikely event of fire, several pumps at the base can send up 500 gallons of water per minute. There is also a reservoir of water just above the main pod. Lightning strikes the tower about 60 times per year but there's probably no safer place during a storm. Every surface which could possibly attract lightning is attached to three copper strips running down the tower, connected to forty-two 22 foot grounding rods buried 20 feet below the surface. Since the tower opened in '78, Prof. Wasyl Janischewskyj of the University of Toronto has been in charge of a lightning study at the CN Tower. They've found that lightning starts from the smaller surface. So if it hits a lake or a field, the lightning indeed comes down from the clouds. If the lower end of the path is the tip of a tall building, the clouds are then the larger surface and the lightning goes up. Since lightning is the result of opposite static charges between the clouds and ground, any tower reduces the distance the spark is required to jump, which means the discharge occurs at lower voltages. Prof. Janischewskyj's team felt this might do less damage, but then discovered that what appears to be a single bolt is often actually several rapid flashes when the lower end is the tip of a tall structure. Information from the study will undoubtedly assist power companies design insulators on hydro towers that are less likely to short-circuit. Special thanks to Joan Cormier and Penny Wright of the CN Tower for supplying the photo and these interesting facts. Extra special thanks to Patricia Kelly, and to Allan Stokell of Positive Images, for helping us find the supplier.

Transactor
The Magazine for Commodore Programmers

Editor-in-Chief
Karl J. H. Hildon

Publisher
Richard Evers

Technical Editor
Chris Zamara

Submissions Editor
Nick Sullivan

Customer Service
Jennifer Reddy

Production
Attic Typesetting Ltd.

Printing
Printed in Canada by
MacLean Hunter Printing

Program Listings In Transactor

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

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

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

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

Cursor Characters For PET / CBM / VIC / 64

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

Colour Characters For VIC / 64

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

Function Keys For VIC / 64

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

Please Note: Transactor's phone number is: (416) 764-5273

CompuServe Accounts

Contact us anytime on GO CBMPRG, GO CBMCOM, or EasyPlex at:

Karl J.H. Hildon	76703,4242
Chris Zamara	76703,4245
Nick Sullivan	76703,4353

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

Subscriptions:
Canada $19 Cdn. U.S.A. $15 US. All other $21 US.
Air Mail (Overseas only) $40 US. ($4.15 postage/issue)

Send all subscriptions to: Transactor, Subscriptions Department, 85 West Wilmot Street, Unit 10, Richmond Hill, Ontario, Canada, L4B 1K7, 416 764 5273. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to our Richmond Hill HQ. For best results, use postage paid card at center of magazine.

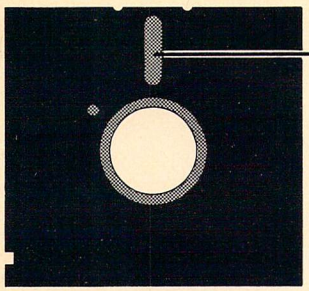
Quantity Orders:

Norland Communications
251 Nipissing Road, Unit 3
Milton, Ontario
L9T 4Z5
416 876 4774

SOLD OUT: The Best of The Transactor Volumes 1 & 2 & 3; Vol 4 Issues 03, 04, 05, 06, and Vol 5 Issues 02, 03, 04 are available on microfiche only
Still Available: Vol. 4: 01, 02. Vol. 5: 01, 04, 05, 06. Vol. 6: 01, 02, 03, 04, 05, 06. Vol. 7: 01, 02, 03, 04, 05, 06. Vol. 8: 01, 02, 03

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

Editorial contributions are always welcome. Minimum remuneration is $40 per printed page. Preferred media are 1541, 2031, 4040, 8050, or 8250 diskettes with WordPro, WordCraft, Superscript, or SEQ text files, or Amiga format 3½ diskettes with ASCII text files. Program listings of more than a few lines should be provided on disk. 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.



Start Address

Copy Copy Revisited

Here are some excerpts from page 3 exactly one year ago:

> *We've been receiving a number of letters regarding local duplication of Transactor Disks.*

Once again the question of duplicating Transactor disks has been the topic for discussion around the office. It's amazing how a problem hits harder at home. Software piracy has been all around us for years and although you know it's affecting the industry, you can't imagine the impact until you become a victim. Recently we've learned that many user groups and other "vendors" have been offering Transactor disks on a regular basis at substantially less than our retail price.

We're not to trying to find fault or place blame. In fact, we probably have no one to blame but ourselves. Although our disk labels show a copyright notice, up until this issue we stated right on our policies page (page 2) that our programs are "public domain; free to copy, not to sell". This notice goes back about 4 years – a popular phrase originally designed to prevent one's program from being "acquired" by someone in the software business. However, the fee at regular club meetings to obtain a copy of any disk being offered that night is carefully termed a "copying fee". Usually it's about $5.00. And 5 bucks isn't a lot – there's equipment wear and tear, time, trouble and transportation, site rental perhaps, plus any number of expenses a club might have to dole out before making the first copy of the evening. Clubs can claim they're not "selling" Transactor disks, but that's only bending the original intent of our policy, which was to avoid the idea that we wanted to exert totalitarian authority over the personal possession of any machine-readable program from Transactor magazine.

> *. . .what little profit we take doesn't even put a dint in our total expenses.*

So if our programs are public domain, what's the difference between our disks and any other club disk? Well, it was an error for us to use the term "public domain" at all – our programs are copyright, as stated on every label of every disk we sell. The programs we publish aren't just donated – we **pay** for the right to print an author's work. . . a total of almost $3000 every issue. Beyond that, we invest hundreds of hours in producing every T. disk master. We just can't *afford* to release the disks to the public domain.

> *All we ask is that you "de-unify" the disks into their respective categories. . . the unified collection will cost less.*

Which is what we sell. Unfortunately, it appears that this policy too has been somewhat more broadly interpreted than we intended. As you may know, the programs in the last 20 issues have been chosen from our submissions to fit a predetermined theme. Effectively, then, the main programs from any issue were already in a "respective category". The disks for these magazines always start with a few utility programs, followed by our short Bits, and then our main program listings. Separating just the main programs onto another disk, with perhaps some other programs of the same nature, was essentially respecting our request. It's difficult to say whether the result was a significant loss in sales, but if so it was once again our own fault.

> *. . .manufacturers need to offer more in a package than just a program to accrue sales.*

Unlike a software firm that can protect against piracy by offering documentation and other services to registered users only, support for our programs is easily obtained just by dropping the subscription card in the mailbox. You even get updates! So, maybe our new disk labels will help. That's right, we've decided to take some of our own advice. In an attempt to protect against piracy (without protecting the disks) we're going to offer something that can only be obtained from the manufacturer. Starting with the disk for this issue, all Transactor disks will be shipped with two-colour labels that should distinguish them from any other disk in your collection. They'll be colour–coded to the cover of the corresponding magazine, so if the logo is green, as on this issue, the disk label will also be green. You might say, "so what". Well, another feature that I'm really looking forward to is having the entire directory printed right on the label! We've selected a label size that will accommodate the whole directory listing, and although the file names will be small, I intend to push this typesetting equipment to the limit to ensure their legibility.

Back on the duplication issue, we've composed a Site Licensing agreement for anyone wishing to make multiple duplicates of Transactor disks. Simply send us $3.00 for every copy made. Including this surcharge, club members will still pay less than our retail price, club treasuries will still continue to benefit from the sales, and the clubs will still be able to offer the service value of on-the-spot availability.

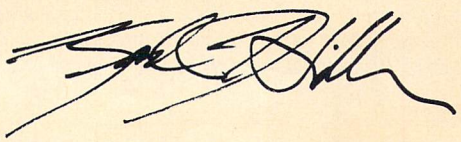
Will the member need to produce his or her copy of the magazine before they're allowed their copy? No. But, they should be aware that unlike games or a menu driven application, most of the programs we publish are of limited value without the documentation in the magazine.

> *Remember, we wouldn't be the first computer publication to become another page in Chapter 11. . .*

What will we do if the clubs and other offending parties don't comply? Probably nothing. But remember, if your club or anyone else sells or allows copying of Transactor disks, they are competing against us with our own product! Although we'd rather be the only source, we'd much rather stay in business. Our Site Licensing agreement is a fair compromise that we hope will be adopted and upheld by interested parties.

A package is going out within the next week to over 400 clubs around the world. The club addresses were obtained from the list published regularly in Commodore magazine, so if your club isn't on this list, please contact us. The package contains most of the above information, plus details of our 20/20 deal. As described in News BRK, the 20/20 deal means 20% off for any order of 20 subscriptions, disks, or any other Transactor product. This is a natural for Commodore user groups, but you don't need to be a club member to take advantage of it. So post the notice on your local BBS – if 19 others respond, the total savings are pretty respectable! The area school board office is another good place to spread this announcement.

On a lighter note, what do you think of our new cover design? (Duke's in the Skypod). It isn't necessarily permanent – we're just experimenting with it while we're off the newsstand. Comments or suggestions, anyone? Now's the time to put in your two cents' worth. Oh yes, one more thing: one free Bits book goes to the first long distance caller to say the word "shazam", and another to the 10th local caller.



Karl J.H. Hildon, Editor in Chief

Using "VERIFIZER"

The Transactor's Foolproof Program Entry Method

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

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

SYS 634 to enable the PET/CBM version (off: SYS 637)
SYS 828 to enable the C64/VIC version (off: SYS 831)
SYS 4096 to enable the Plus 4 version (off: SYS 4099)
SYS 3072,1 to enable the C128 version (off: SYS 3072,0)
BANK 15: SYS 1024 for B128 (off: BANK 15: SYS 1027)

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

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

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

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

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

PET/CBM VERIFIZER (BASIC 2.0 or 4.0)

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

VIC/C64 VERIFIZER

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

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

When using 'VERIFIZER' with some TVs, the upper left corner of the screen is cut off, hiding the verifizer-displayed codes. DOUBLE VERIFIZER solves that problem by showing the two-letter verifizer code on both the first and second row of the TV screen. Just run the below program once the regular Verifizer is activated.

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

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

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

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

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

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

Plus 4 VERIFIZER

```
NI 1000 rem * data loader for ' 'verifizer +4' '
PM 1010 rem * commodore plus/4 version
EE 1020 graphic 1: scnclr: graphic 0: rem make room for code
NH 1030 cs=0
JI 1040 for j=4096 to 4216: read x: poke j,x: ch=ch+x: next
AP 1050 if ch<>13146 then print ' 'checksum error' ': stop
NP 1060 print ' 'sys 4096: rem to enable' '
JC 1070 print ' 'sys 4099: rem to disable' '
ID 1080 end
PL 1090 data  76,  14,  16, 165, 211, 141,   2,   3
CA 1100 data 165, 212, 141,   3,   3,  96, 173,   3
OD 1110 data   3, 201,  16, 240,  17, 133, 212, 173
LP 1120 data   2,   3, 133, 211, 169,  39, 141,   2
EK 1130 data   3, 169,  16, 141,   3,   3,  96, 165
DI 1140 data  20, 133, 208, 162,   0, 160,   0, 189
LK 1150 data   0,   2, 201,  48, 144,   7, 201,  58
GJ 1160 data 176,   3, 232, 208, 242, 189,   0,   2
DN 1170 data 240,  22, 201,  32, 240,  15, 133, 210
GJ 1180 data 200, 152,  41,   3, 133, 209,  32, 113
CB 1190 data  16, 198, 209,  16, 249, 232, 208, 229
CB 1200 data 165, 208,  41,  15,  24, 105, 193, 141
PE 1210 data   0,  12, 165, 208,  74,  74,  74,  74
DO 1220 data  24, 105, 193, 141,   1,  12, 108, 211
BA 1230 data   0, 165, 210,  24, 101, 208, 133, 208
BG 1240 data  96
```

C128 VERIFIZER (40 column mode)

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

B128 VERIFIZER **Elizabeth Deal, Malvern, PA**

```
1 rem save' '@0:verifizerb128' ',8
10 rem* data loader for ' 'verifizer b128' ' *
20 cs=0
30 bank 15:for i=1024 to 1163:read a:poke i,a
40 cs=cs+a:next i
50 if cs<>16828 then print' '** data error **' ': end
60 rem bank 15: sys 1024
70 end
1000 data  76,  14,   4, 165, 251, 141, 130,   2, 165, 252
1010 data 141, 131,   2,  96, 173, 130,   2, 201,  39, 240
1020 data  17, 133, 251, 173, 131,   2, 133, 252, 169,  39
1030 data 141, 130,   2, 169,   4, 141, 131,   2,  96, 165
1040 data   1,  72, 162,   1, 134,   1, 202, 165,  27, 133
1050 data 233,  32, 118,   4, 234, 177, 136, 240,  22, 201
1060 data  32, 240,  15, 133, 235, 232, 138,  41,   3, 133
1070 data 234,  32, 110,   4, 198, 234,  16, 249, 200, 208
1080 data 230, 165, 233,  41,  15,  24, 105, 193, 141,   0
1090 data 208, 165, 233,  74,  74,  74,  74,  24, 105, 193
1100 data 141,   1, 208,  24, 104, 133,   1, 108, 251,   0
1110 data 165, 235,  24, 101, 233, 133, 233,  96, 165, 136
1120 data 164, 137, 133, 133, 132, 134,  32,  38, 186,  24
1130 data  32,  78, 141, 165, 133,  56, 229, 136, 168,  96
1140 data 170, 170, 170, 170
```

b i t s

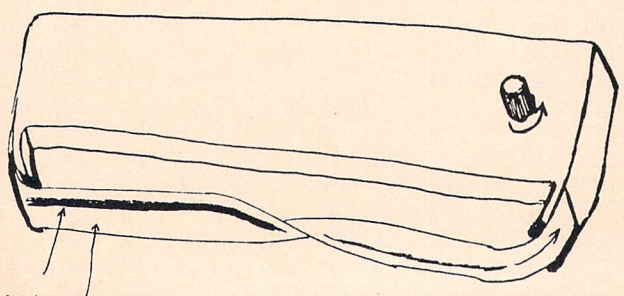
Got an interesting programming tip, short routine, or an unknown bit of Commodore trivia? Send it in – if we use it in the Bits column, we'll credit you in the column and send you a free one–year's subscription to The Transactor

Ribbon Rejuvenation

Murray Kalisher
Santa Barbara, CA

Most of today's popular dot matrix printers use easily–replaceable ribbon cartridges. These usually contain a long continuous loop of nylon ribbon folded and "crunched" inside a plastic cartridge. One short section of the ribbon is exposed to the dot matrix print head and the ribbon is moved in one direction continuously as printing occurs. If you remove a worn cartridge you may notice that usually only the upper half of the ribbon has been used. Certain types of cartridges allow increased ribbon life by "flipping" the ribbon internally so that both top and bottom halves are used. For cartridges where this is not the case, you can do it yourself without too much trouble, and double your ribbon life.

First notice the direction in which the ribbon travels by turning the winding knob located at the top of the plastic case. Loosen the ribbon a bit by pulling on it and give it a half twist to reverse the top and bottom edges. Place the twisted section of ribbon over the plastic guide closest to where the ribbon enters the cartridge and turn the knob to remove the slack in the ribbon. Now keep turning the knob to pull half of the "twist" into the cartridge. Once it's inside you just keep turning the knob for perhaps 10 or 15 minutes until the twist comes out the other end of the cartridge and the ribbon is completely untwisted again. Now notice that the unused half of the ribbon is at the top edge. You now have essentially a "new" ribbon!



Used 1/2
Unused 1/2

This trick will not work with carbon film type ribbons because the carbon will be on the wrong side of the film if you twist it. I tried this trick with the ribbon for my Citizen printer, which is a standard Epson MX–80 ribbon, and it works great!

CAUTION!

Dean Rouncville
Athabasca, Alberta

There is an easy way to destroy a BASIC program in memory, one that I found out in a most unpleasant manner. When entering a line in a long program, there is a short delay before the cursor comes back, while the interpreter re–chains the program line links. On very long programs, this delay can be several seconds long. If you interrupt this process by hitting RUNSTOP/RESTORE before the cursor comes back, your program will be in quite a mess, as the rechaining process was not completed. Try LISTing a program after a RESTORE in such a circumstance and you'll see. Sometimes you can get out of the mess just by re–typing the line you just entered; other times, repairing the damage could be considerably more difficult than that.

I hope this warning will save someone the agony of losing two hours of programming time, as I did.

Defaults

Amir Michail
Willowdale, Ontario

The need to save default data for a program arises many times. To do it, one need not set up a sequential file to store the default data. Here is an elegant solution which provides 255 bytes of data storage (enough for 50 top scores in a game!). As a bonus, programs may be loaded and then saved on another disk – the defaults will automatically be saved with the program!

First a little theory. When you load a program, the last track and sector used is stored at 24–25 ($18–$19) in the 1541. The idea is to retrieve this information from the drive for later use, and load in the default data from that sector.

Since that last sector has just been loaded, then one need not tell the drive to read it again – all the information is there in the drive's buffer that was just used! However, the track and sector should be read from drive memory since the user may wish to change the default data later on.

A demonstration program follows that clearly shows how this technique can be used. However, before you run it you must add a data block. First type in the program and save it as "demo". Next type in the following in direct mode:

```
open5,8,5,"demo,p,a":forj=1to254
:print#5,chr$(j);:next:close5
```

Now load the program and run it! Warning: if you want to edit the program, remember to create another data block.

```
LP  10 rem* default demo -- july 1987
GE  20 rem* by amir michail
CA  30 rem* saves and reads default values
EO  40 rem* from last block of program
GK  50 :
PI  100 z$=chr$(0): open 15,8,15
GJ  110 open 2,8,2,"#"
FE  120 rem read values from disk buffer
GA  130 print#15,"m-r"chr$(24)chr$(0)chr$(2)
HJ  140 get#15,t$,s$:t=asc(t$+z$):s=asc(s$+z$)
DM  150 print#15,"b-p";2;2
FK  170 get#2,c1$,c2$,c3$
GJ  180 poke53281,asc(c1$+z$):poke53280,
        asc(c2$+z$):poke646,asc(c3$+z$)
CC  195 close2: close15
MD  210 print ,"change color defaults";
DO  220 input k$:if k$="n" then 320
MG  230 input "background, border, text colors:";
        c1,c2,c3
EE  240 poke53281,c1: poke53280,c2: poke646,c3
KB  250 open 15,8,15: open2,8,2,"#"
PA  260 rem write values to last sector
LD  270 print#15,"b-p";2;2
GP  280 print#2,chr$(c1)chr$(c2)chr$(c3);
LO  290 print#15,"u2";2;0;t;s
ED  300 print "done!"
FJ  310 close2: close15
AL  320 print "program can start from this point."
KE  330 end
```

Peek-a-Page — **Hardy Moore, Bridgewater, MA**

One way to read a page from disk memory is to enclose a memory read and a get# within a loop and execute it 256 times. Here is a quicker approach:

```
10 open 15,8,15
20 print#15,"m-r"chr$(lo)chr$(hi)chr$(0)
30 for i= 0 to 255
40 get#15,b$
50 print i;asc(b$+chr$(0))
60 next i
```

The first two parameters of the memory-read command specify the starting address in drive memory, and the third parameter is the number of bytes to read. Although zero seems like a curious value to use, it actually causes the drive to supply 256 bytes due to the counting algorithm used by DOS.

Since the "m-r" command precedes the loop, there are 255 fewer commands to send across the serial bus and have the drive execute.

File Hider — **Michael Bone, Strathalbyn, South Australia**

Here's an easy way to hide files so that they won't show up in a directory. This short program will hide every file in the directory starting from the one you specify. You can still load the "invisible" files, and you can load the first hidden file using the filename "??blocks free*".

```
10 input "hide files from";a$
20 open 1,8,15
30 print#1,"r0:"+chr$(20)+chr$(20)+"blocks free"
   +chr$(0)+chr$(0)+chr$(0)+"="+a$
40 close 1
```

To get the directory back to normal, just rename the file back again, like this:

```
print#1,"r0:oldname="+chr$(20)+chr$(20)
+"blocks free"+chr$(0)+chr$(0)+chr$(0)
```

Basic Bugs

More on the VAL Bug

In Volume 8 Issue 1, we reported on an "Obscure C-64 VAL bug" and the problems it could cause with interrupt-driven programs. In a nutshell, it happened like this: the VAL function would put a zero at the end of the string being evaluated, call a subroutine to evaluate the null-terminated string, then replace the zero with what was there originally. The bug was that an interrupt routine starting right above string storage would get stepped on if a VAL was performed on the first string created in a BASIC program, and bomb out if it was executed while the VAL was being performed.

Recently, Larry Phillips of Vancouver, BC, brought another manifestation of this bug to our attention, one that is easier to reproduce. For a dramatic demonstration, try entering this program:

```
10 a$ = "4e99": print val(a$): rem this will disappear
20 rem but this won't
```

RUN the program (you'll get the message "overflow error in 10"), then LIST it. You will notice that BASIC took the liberty of munching part of line 10, as predicted by the REM statement. Everything after the "4e99", including the closing quote, is wiped out.

The reason lies in another, related, bug in VAL. If an error occurs in the evaluation of the string, like the overflow error in this case, the zero that was stored at the end of the string is never replaced by the original contents of that location. Since a string defined in a BASIC program exists as a pointer to the program text itself, VAL is actually putting that zero right smack into your BASIC code! If VAL bombs, you've got a line-terminating zero sitting in your program to make things mysteriously disappear.

This bug won't often be a problem, but it is interesting to note that a seemingly innocent-looking bit of code can be self-destructive. The same problem will not occur on a C128, because the string does not exist as a pointer to BASIC code, but is copied into string storage in high memory along with the dynamically-defined strings.

Multiply Bug

Dave Dixon
Vancouver, BC

Here's another BASIC bug that shows up on the 64, but is fixed on the 128. This one is in the multiply routine. Try this:

```
print 8388608.88 * 1
print 1 * 8388608.88
```

And see what you get.

LIST During a Program

Jason Dorie
Ft. McMurray, Alberta

As you may know, a LIST command in a program will end it right after the LIST is finished. This line takes care of that:

```
10 poke 768,174: poke 769,167: list
 : poke 768,139: poke 769,227
```

Line ranges may be specified with the LIST command as usual.

Inside View

Scott Gray
New Bloomfield, MO

Have you ever wondered what your computer might see from the other side of the screen? Well, this program will show you just that – it displays the screen just as if you were looking at it from behind. What's more, you can program and otherwise operate your 64 while enjoying this view; any BASIC program that does not use machine language subroutines or an alternate character set will work fine with Inside View, though it will be slowed down a bit.

This program really has no practical value; it just serves to demonstrate what weird and wonderful things the C64 is capable of.

It's easy to toggle between the normal screen and the reversed screen:

```
normal  : POKE 53272,21: POKE 56576,151
reverse : POKE 53272,37: POKE 56576,148
```

Have fun!

```
LO 1 rem ******************************
GE 2 rem *                            *
NB 3 rem *      inside view by scott gray *
IE 4 rem *                            *
JM 5 rem *    the view that your computer *
OI 6 rem *    sees from the other side of *
HN 7 rem *          the screen        *
ME 8 rem *                            *
DP 9 rem ******************************
GB 10 for x=49152 to 49466:read a
EA 20 poke x,a:c=c+a:next
HF 30 if c<>43135 then print "error!":stop
NH 40 sys 49152:end
JP 1000 data 169, 000, 141, 014, 220, 133, 251
NB 1001 data 133, 253, 168, 169, 051, 133, 001
OC 1002 data 169, 048, 133, 252, 169, 208, 133
KA 1003 data 254, 177, 253, 132, 002, 133, 003
LM 1004 data 160, 007, 006, 003, 102, 004, 136
JA 1005 data 016, 249, 165, 004, 164, 002, 145
HA 1006 data 251, 200, 208, 232, 230, 252, 230
IB 1007 data 254, 165, 254, 201, 224, 208, 222
PA 1008 data 169, 055, 133, 001, 169, 001, 141
FP 1009 data 014, 220, 120, 165, 001, 041, 251
OP 1010 data 133, 001, 169, 000, 133, 251, 133
HF 1011 data 253, 168, 169, 048, 133, 252, 169
OD 1012 data 208, 133, 254, 177, 251, 145, 253
MB 1013 data 200, 208, 249, 230, 252, 230, 254
MC 1014 data 165, 254, 201, 224, 208, 239, 165
KP 1015 data 001, 009, 004, 133, 001, 088, 169
HB 1016 data 148, 141, 000, 221, 169, 037, 141
HA 1017 data 024, 208, 120, 169, 134, 141, 020
IC 1018 data 003, 169, 192, 141, 021, 003, 088
LD 1019 data 096, 169, 004, 133, 252, 169, 200
HC 1020 data 133, 254, 169, 000, 133, 253, 133
PC 1021 data 251, 168, 162, 039, 185, 000, 004
CB 1022 data 157, 000, 200, 185, 040, 004, 157
FD 1023 data 040, 200, 185, 080, 004, 157, 080
CA 1024 data 200, 185, 120, 004, 157, 120, 200
FE 1025 data 185, 160, 004, 157, 160, 200, 185
KA 1026 data 200, 004, 157, 200, 200, 185, 240
AB 1027 data 004, 157, 240, 200, 185, 024, 005
CD 1028 data 157, 024, 201, 185, 064, 005, 157
HB 1029 data 064, 201, 185, 104, 005, 157, 104
DC 1030 data 201, 185, 144, 005, 157, 144, 201
FG 1031 data 185, 184, 005, 157, 184, 201, 185
MB 1032 data 224, 005, 157, 224, 201, 185, 008
JB 1033 data 006, 157, 008, 202, 185, 048, 006
```

```
CF  1034 data 157, 048, 202, 185, 088, 006, 157
DE  1035 data 088, 202, 185, 128, 006, 157, 128
EE  1036 data 202, 185, 168, 006, 157, 168, 202
LD  1037 data 185, 208, 006, 157, 208, 202, 185
CE  1038 data 248, 006, 157, 248, 202, 185, 032
OC  1039 data 007, 157, 032, 203, 185, 072, 007
BE  1040 data 157, 072, 203, 185, 112, 007, 157
ID  1041 data 112, 203, 185, 152, 007, 157, 152
JF  1042 data 203, 185, 192, 007, 157, 192, 203
LC  1043 data 200, 202, 192, 040, 240, 003, 076
AG  1044 data 151, 192, 076, 049, 234, 078, 000
```

Mobile BASIC

Mark Schreiner
Overland Park, KS

The short program below enables you to move the BASIC area anywhere in the Commodore 64's memory. Answer the prompts with the starting and ending addresses of BASIC's new location and the machine will appear to be reset but the "BYTES FREE" message will reveal that you are in your new zone.

Moving BASIC opens up some interesting possibilities. Try 828–1023 (the cassette buffer) and 49152–53247 (the free memory not contiguous with the normal BASIC workspace). You can now load in machine language programs and other files (using the ",8,1" non–relocating load option) that load into the normal BASIC area. With BASIC moved, you may use a BASIC program to examine the contents of the normal BASIC area.

Make sure you save this program before running it – when you move BASIC, the program won't exist in the new area and you'll lose it. Just re–load the program in the new area if you want to re–locate again.

```
MA  10 for i=0 to 18: read a: poke 828+i,a: next
NI  20 input"start";bs: poke 831,int(bs/256)
OH  30 poke 829,bs-(int(bs/256)*256)
DD  40 input"end";be: poke 839,int(be/256)
OF  50 poke 837,be-(int(be/256)*256):sys828
CE  60 data 162, 0, 160, 0, 24, 32, 156, 255
DE  70 data 162, 0, 160, 0, 24, 32, 153, 255
GK  80 data 108, 0, 160
```

Bytes Free

George Borsuk
Brantford, Ontario

Here's a short, relocatable machine language routine to display the number of BASIC bytes free. Unlike BASIC's FRE function, this routine always prints the number of bytes free properly, as a positive number. It is based on a ROM routine that is executed as part of the power–up sequence.

It subtracts the bottom–of–memory pointer at $2D (top of your program) from the top–of–memory pointer at $37, then uses the ROM routine at $BDCD to print the resulting value. Finally, the "BASIC BYTES FREE" message stored at $E460 is printed by the print string routine at $AB1E.

You can use this program as–is by just SYSing to it from BASIC, or include the code in your own ML programs. Both BASIC loader and assembler code are listed below:

```
 1 rem* bytes free program *
 2 rem* by george borsuk   *
10 for i=49152 to 49172
20 read a: poke i,a: ch=ch+a
30 next i
40 if ch<>2624 then print"data error!!!": stop
50 print"** sys 49152 will give bytes free"
60 data 165,  55,  56, 229,  45, 170, 165
70 data  56, 229,  46,  32, 205, 189, 169
80 data  96, 160, 228,  32,  30, 171,  96
```

The machine code:

```
lda $37     ;top of memory, low
sec
sbc $2d     ;subtract bottom, low
tax
lda $38     ;memtop high
sbc $2e     ;subtract bottom high
jsr $bdcd   ;print number in a,x
lda #$60    ;point to string
ldy #$e4    ; at $e460
jsr $ab1e   ;print message
rts         ;return to BASIC or caller
```

Getting The Boot

Chris Miller
Kitchener, Ontario

Ever wish you could load in machine language or data files into your C64 without the danger of crashing out of BASIC with a FILE NOT FOUND error, or having your program re–run from the top after the load? All it takes is a couple of SYS calls and a poke!

```
10 rem  boot to load 3 programs from basic
20 sn = 57812:  rem set name
30 ld = 65493:  rem kernal load
40 a  = 780 :   rem .a register params
50 sys sn"prog1",8,1: poke a,0: sys ld
60 sys sn"prog2",8,1: poke a,0: sys ld
70 sys sn"prog3",8,1: poke a,0: sys ld
80 rem continue with basic
```

The flow of BASIC is not disturbed by these loads and if an error occurs, you can handle it yourself without the program bombing. Just check the disk error after each load with something like:

```
open 15,8,15: input#15,a,b$,c,d: close 15
 : if a then print b$ (error)
```

Use this technique for ". . . ,8,1" loads only.

You can also use this in direct mode to load code or data without disturbing BASIC's start–of–variables pointer.

128 Bits

Booting Up Your Mode Menu

John Chism
Knoxville, TN

Puttering around the C128 Programmer's Reference Guide, I found some data on page 447 that, when entered into the boot sector (Track 1 sector 0) of a disk, purported to provide the user, upon booting, with a mode menu for:

1) C64 BASIC
2) C128 BASIC
3) C128 MONITOR

This is not quite so. The ML routine should begin on the 9th byte, not the 12th, and the loop waiting for input is not properly executed.

The following program is a properly working version, and writes the code to the boot sector of a disk. But be forewarned that it should be installed on a fresh disk, or on one that you know doesn't have track 01, sector 00 allocated.

Just run this program on the fresh disk to install the boot menu. In the future, whenever you boot with that disk in the drive, the above menu will come up and allow you to make a selection.

```
BN 100 rem* select on boot
DO 110 rem* fix for pg. 447 of c128 prg. guide
HE 120 rem* track 1 sector 0 must not be allocated
NC 140 for i=1 to 100
OG 150 read byt: cs=cs+byt: w$=w$+chr$(byt)
EK 160 next
AP 170 if cs<>6965 then print "data error": end
IM 180 open 15,8,15,"i0"
KA 190 open 8,8,8,"#"
JA 200 print#15,"b-p";8;0
JP 210 print#8,w$
LC 220 print#15,"u2";8;0;1;0
LG 230 print#15,"b-a";0;1;0
DH 240 print ds$: close 8: close 15
BP 250 data  67,  66,  77,   0,   0,   0,   0,   0
NI 260 data   0,  32, 125, 255,  13,  83,  69,  76
CL 270 data  69,  67,  84,  32,  77,  79,  68,  69
FH 280 data  58,  13,  13,  32,  49,  46,  32,  67
GH 290 data  54,  52,  32,  32,  66,  65,  83,  73
LH 300 data  67,  13,  32,  50,  46,  32,  67,  49
PI 310 data  50,  56,  32,  66,  65,  83,  73,  67
CH 320 data  13,  32,  51,  46,  32,  67,  49,  50
GN 330 data  56,  32,  77,  79,  78,  73,  84,  79
CF 340 data  82,  13,  13,   0,  32, 228, 255, 201
CP 350 data  49, 208,   3,  76,  77, 255, 201,  50
MM 360 data 208,   3,  76,   3,  64, 201,  51, 208
LF 370 data 235,  76,   0, 176
```

Quick File Copier

T.A. Sweeney
Ridgefield, CT

Here's a short program to copy program files from one disk to another. It simply automates a BLOAD/BSAVE process, allowing program files to be copied at a high speed while maintaining their original load address. It was written in response to problems I had with the file copy feature of the DOS shell.

```
MN 100 rem  program file copier 128  tim sweeney
CH 110 poke 58,5   :rem reserve space in bank 1
AL 120 clr
GC 130 print"prg file copier"
FM 140 print"insert master disk, press key": getkey a$
IO 150 input"file name";fl$: dclear
AK 160 bload (fl$),b1
PH 170 sa=peek(172)+peek(173)*256 :rem  get
       starting address
IM 180 ea=peek(174)+peek(175)*256 :rem  get
       ending address
KP 190 print"insert target disk, press key": getkey a$
IF 200 dclear: bsave (fl$),b1,p(sa) to p(ea)
FD 210 print"copy completed. another? y";
       : input"[3 left]";a$
NC 220 if a$="y" then 120
DG 230 poke 58,255: clr   :rem  restore bank 1
```

FAST GULP

M. Garamszeghy
Toronto, Ontario

With a few simple modifications, the GULP.COPY program presented in Transactor Vol. 7, Issue 06 (page 52) for the 1571 and C-128 (1571 RAM disk copier), can be doubled in speed. The modifications described below allow the program to copy a single sided disk in just over four minutes and a double sided in about seven and a half. The modifications to the main BASIC listing on page 53 are as follows:

```
1121 slow: open 9,8,9,"#3"

1181 print#15,"u0"chr$(8)chr$(4)

1261 print#15,"u0"chr$(8)chr$(4)
1262 restore 2000
1263 of$="": for fx=1 to 29: read of: of$=of$+chr$(of)
       : next
1264 print#15,"m-w"chr$(0)chr$(6)chr$(29)of$
1265 print#15,"m-e"chr$(0)chr$(6)

1330 print chr$(147): print "**done**": print#15,"uj": dclose

2000 data 120, 169,  13, 141, 169,   2, 169,   6
2010 data 141, 170,   2,  88,  96,  72, 165,   0
2020 data 201, 160, 208,   4,  69,   0, 133,   0
2030 data 104,  76, 222, 157,   0,   0
```

The reasons for the changes are as follows:

(1) line 1121 forces the C-128 into slow mode (because direct memory access will not work properly if the C-128 is in FAST mode) and reserves direct access disk buffer #3.

(2) lines 1181 and 1261 use the burst mode "set sector interleave" command to an optimum interleave of 4. This speeds up both disk reads and writes.

(3) lines 1262 to 1265 set up and execute a short program in the 1571's buffer #3. This program turns off the verify after write feature by patching into the 1571's interrupt routine allowing much greater write speeds. (Each time the 1571 writes a sector, it immediately reads it back in again to verify it. This causes a time delay of one disk revolution or about 0.3 seconds of wasted time for each sector written).

(4) The print#15,"uj" in line 1330 replaces an "i0" in the older version. The "uj" will reset the default interrupt vectors in the 1571 after the copying has finished.

(5) lines 2000 to 2030 contain the object code for the 1571 program. The source code is as follows:

```
        sei             ;disable interrupts
        lda  #<writeoff ;low byte of new interrupt routine
        sta  $02a9      ;save in interrupt vector lo byte
        lda  #>writeoff ;new hi byte
        sta  $02aa      ;and save also
        cli             ;restore interrupts
        rts
writeoff  =*            ;new interrupt handler
        pha             ;save accumulator
        lda  $00        ;get job code for buffer#0
        cmp #$a0        ;check for "verify sector"
        bne  continue   ;no, then go back to normal
        eor  $00        ;yes, then cancel it
        sta  $00        ;and save again
continue =*
        pla             ;retrieve accumulator, and
        jmp  $9dde      ;go to normal interrupt handler
```

This short machine code program can be placed in any of the 1571's buffers which are not currently used by DOS. Be careful when using it on your own because failure to reset the interrupt pointer to its default value after use will cause the 1571 to lock up or go hay wire if the new interrupt routine were to become corrupted by being overwritten by DOS.

Built-In Crash Protection

Mike Hartigan
Lockport, NY

A little-known and even less documented feature of the C128 is a built-in way to recover from crashes without losing your BASIC program! If a program locks up, it can generally be saved by holding down the RUN/STOP key while simultaneously pressing the reset button. This will cause the C128 to reset, but instead of initializing BASIC, it will enter the built-in monitor. The interesting part is that zero page has been left virtually untouched, including the pointers to your BASIC program! Now enter X <RETURN> to exit the monitor, and Voila! Your BASIC program is there, right where you left it! This trick will not work if you lost your program to a NEW command, since the pointers have already been reset. In this case, a more traditional un-new technique must be employed.

Common Memory

Tim Fleehart
Tumwater, WA

The C128's MMU (Memory Management Unit) has a register that controls the amount of "shared" or common memory present in both the 128 and Z80 modes. The common memory is always bank 0. This register is located at $FD506 and it works like this:

Binary value of low nybble	Shared memory configuration
00xx	No shared memory
01xx	Share low memory only
10xx	Share high memory only
11xx	Share in both high and low memory
xx00	Share 1k bytes in each section
xx01	Share 4k bytes in each section
xx10	Share 8k bytes in each section
xx11	Share 16k bytes in each section

Example: storing $0F (%00001111) in $FD506 results in 32k of shared memory; 16k from $0000 to $3FFF, and 16k from $C000 to $FFFF (excepting, of course, the configuration memory at $FF00-$FF04).

The purpose of this common memory is to facilitate the stack and the common storage necessary for program 'house-keeping'. The 8502 requires common low memory for a stack and zero

Amiga Bits

Customizing CLI Windows

Bryce Nesbitt
Berkely, CA

When a new CLI is started from the Workbench, the window always comes up in the same, fixed location. If you do not like the position or size you are out of luck. From the CLI a bit more control is possible: a window specification such as "CON:0/1/640/200/Old CLI" can be typed after the NEWCLI command. But that's a lot of typing and is useless for Workbench users. Changing the defaults allows both CLI and Workbench users to get that custom look without all the hassle.

The modification is easy, with one exception: a binary file editor is not part of the software provided to users with their Amiga. The rest of this discussion will assume that you have already

downloaded, or otherwise obtained the program "filezap". The techniques described can also be applied to other editors. (The text editor "Aedit" by Joe Bostic, for example, allows editing of binary files.)

Depending on which CLI startup file you wish to modify, use one of these commands to start filezap:

```
CLI        > run filezap c:newcli
Workbench > run filezap sys:system/cli
```

Now page forward with the <F> command, keeping a sharp lookout for a string that looks something like "CON:0/50/640/80/New CLI". This defines the characteristics of the new CLI window:

"CON:" means open a console window – leave this part alone. The first number is how many pixels from the left edge the window should start. The seconds number is how many pixels from the top. The third is the width in pixels, and the final one is the height. The "New CLI" will be used as the title of the window.

Use filezap's <*> command to change the defaults. Keep in mind that the window must fit on the screen, and that you can't change the total length of the string! If you make a mistake, use the <R> command to recover. When ready, use the <U> command to update the file, and press <CTRL><C> to exit.

Now your CLI window will come up where and how you like it!

Titles in AmigaBasic

John Chen
Clifton, NJ

It's difficult to change the title of a window in AmigaBASIC after the window has been opened, and impossible to name the screen title. This program uses a system library routine to circumvent this BASIC weakness. For the program to work, the "intuition.bmap" file has to be in the "libs:" directory on the Workbench disk or in the same directory the program is stored in. You can get the ".bmap" file from the public domain or, if you have the "Amiga Enhancer kit" (the 1.2 system software upgrade), you can use the "ConvertFD" program on the Extras disk (in the "BasicDemos" directory) to roll your own .bmaps from the files in the "FD1.2" directory.

```
LIBRARY "intuition.library"
ws& = WINDOW(7)  ' pointer to a Window structure
INPUT"Window Title";wt$
INPUT"Screen Title";st$
wt$ = wt$ + CHR$(0)  ' for no title set
st$ = st$ + CHR$(0)  ' string to chr$(0)
CALL SetWindowTitles(ws&,SADD(wt$),SADD(st$))
' SADD returns the address where the string is stored
LIBRARY CLOSE
END
```

Easter Eggs

We've seen computers before with secret messages hidden within their operating systems like Easter eggs: the original PET had Microsoft's "WAIT 6502,x" joke; the C–128 has its "SYS 32800,123,45,6" incantation that puts up credits and a peacenik message. The Amiga takes this trend one step further.

First step: Move any windows out of the way that are covering up the Workbench screen's title bar, and click somewhere on the Workbench window to activate it. You should see the familiar "Workbench release. . . " message with the amount of free memory displayed in the title bar. It is in the title bar that the secret messages will be displayed.

Now, you'll have to hold down several keys at once for this: with the thumb of your left hand, hold down the left SHIFT and ALT keys simultaneously, and with the thumb of your right hand, hold down the right SHIFT and ALT keys. Now, with the middle finger of your left hand, hold down the F1 key – all five keys should be down at this point. Poof! With the flash of a "display beep", up comes a secret message in the title bar! Now release F1 and hold down F2 instead. You get a new message for each of the ten function keys.

If you thought getting those messages was a bit tricky, you'll find that getting to the next stage of messages is like playing solitaire twister. First of all, make sure the Workbench window is activated, the title bar is visible, and the Workbench disk is in the internal disk drive. Your mouse should be placed up against the right side of the keyboard, close to the Amiga. Also, the mouse pointer on the screen must be resting over the screen–to–back re–ordering gadget in the Workbench screen title bar (the second gadget from the right). To get the mouse and pointer in these positions, you may have to lift the mouse up and put it down where you want it. Now you are ready for stage two: first get the F1 secret message displayed as explained above. Now, with all five keys still held down, reach out with the ring finger of your right hand and eject the Workbench disk from the drive. Poof! you will see another secret message appear in the title bar. Now don't move! You are ready for stage three.

Stage three should only be attempted if there are no children present, and you are not offended by naughty words. While in the above position (the mouse pointer should still be over the screen–to–back gadget), reach out to the left mouse button with the little finger of your right hand and hold it down, being careful not to move the mouse. Now, get this: With the mouse button still held down, you have to re–insert the disk! Use your ring finger for this. After the disk "catches", poof! The naughty message! Wonder what they mean?

This isn't a joke; it really works, but it can be tricky to do. You may want to enlist the aid of an assistant to get to stage three. Another way is to use the program in this issue called "Eventmaker" to do the work for you just by typing in a command. The command to get the secret message is given at the end of the article.

Letters

Glossy paper pricing: I love the way you guys work. You start to use more expensive glossy paper rather than the thick non-gloss paper (that is *a lot better* than glossy), raising the expense for materials, then you raise your price for subscriptions. I would think that the extra price for subscribing could go to something more useful than easily-ripped glossy paper (which I hate!). How about more articles and projects. How about more mail order products. If you really need more income, get more ads for non-Transactor products. Then lower your prices for subscriptions and mail order items.

I hope you will think more heavily about the above.

Steven T. Campbell, Mississauga, Ontario

We don't like increasing the price of a subscription any more than you like paying it, Steven (which is why the price didn't change for so long). And strangely enough, we also prefer the old paper, though not everyone agrees with us. But your assumption that the glossy paper is more expensive is incorrect – it's actually about 15 per cent cheaper, which is one reason we changed it. Another reason is that advertising copy looks better on the coated stock – at least in the eyes of the advertisers. Your other points are well taken: we are going after more outside advertising, we will have more mail order products and, as the advertising starts to build, we also want to increase the number of pages per issue to accommodate more articles and projects. Meanwhile, though, we have to stay in business. . . and we hope you'll stick around to watch us grow.

Blazin' Forth docs: Since discovering your magazine in March, I have found it to be an excellent source of technical information on Commodore's computers. I was so impressed by that first issue (More Languages) that I got a disk and magazine subscription, and the companion disk to that issue. Unfortunately I missed the Simulations and Modelling issue, but I haven't been disappointed since.

One article that interested me in the More Languages issue was "Blazin' Forth – Everything you wanted to know about Forth (but were afraid to ask)". I was thrilled to get the Forth system on the disk, but was disappointed to find no documentation on how to *use* it. In the first paragraph of the article, the author stated: "I wrote the following as an aid to people who might be trying to understand the source to Blazin' Forth, and it should be considered part of the documentation for the source files to the system." How can I get the documentation on how to use it and, if possible, the source files as well?

David Neto, Toronto, Ontario

Yours is one of a number of inquiries we have received about more docs for Blazin' Forth, David. Unfortunately, the source and documentation for Scott Ballantyne's superb implementation of Forth are just too large to distribute ourselves. They are available, though, on the Commodore programming (CBMPRG) forum on CompuServe. If you have a CompuServe account (or if you were to get one) you could also put questions to Scott Ballantyne on-line, if you desire, as he checks in on our forums fairly regularly. If you (or anyone for that matter) would like a free CompuServe Intro Pak, please call us. It contains an account number, a password, and $15 of on-line time.

Drive head noise abatement: In a letter in your September issue, Warren Pollans asked for solutions to the problem of head banging on the destination drive when using fast dual-drive copy program (Letters, Volume 8, Issue 2).

I overcome this problem when using Fast Hack'em by pressing F1, which changes the source from device 8 to device 9; then I call for a directory (by pressing 'D') with the source disk in device 9. The head does not bang during this operation, and the head position is now known, so copying can proceed without fear of damage to the head alignment. While on the subject of this particular program, using the dual file copiers in Fast Hack'em (v3.99) seems to be

impossible between a 1571 (even in 1541 mode) and a 1541. Solutions, anyone?

Bruce Lloyd, Dapto, New South Wales

The 68010 and commercial disks: The article on the 68010 Amiga (The 32-bit Amiga, Volume 8, Issue 2) raised some concern. Most purchased software is loaded following Kickstart and has its own startup-sequence. As I understand it, if one goes for a conversion one is faced with modifying the start-up sequence of all owned software and copying DeciGEL to all disks. Is there any way around this problem?

Ian Robertson, Belleville, Ontario

Interesting point, Ian. No, we don't know of any way to make DeciGEL active at boot-up without altering the startup-sequence. Don't go changing all your disks, though. . . remember that only source of compatibility problems when you install the 68010 is the comparatively rare MOVE SR,<EA> instruction. Chances are that most of the programs you use do not contain this instruction, so installing DeciGEL won't be required anyway (the worst that will happen, by the way, is that you'll get a Software Error caused by the 68010 detecting what, for it, is a privilege mode violation). Also, many commercial programs can in fact be loaded from either CLI or Workbench without having to reboot, so you can install DeciGEL before invoking them. If necessary, copy the ASSIGN and EXECUTE commands to RAM: (or put your Workbench disk in your external drive if you have one), put the commercial disk in the internal drive, and execute a script file like the following:

```
assign sys: df0:
assign s: df0:s
assign libs: df0:libs
assign l: df0:l
assign devs: df0:devs
assign fonts: df0:fonts
assign c: df0:c
```

Now execute the startup-sequence on the program disk, and chances are the program will load successfully. Failing that, you will have to change the program disk, but that should be required only rarely.

Taking Amiga to task: You have been lamenting the supposition that the Amiga is not selling. I take this as figurative rather than literal, leaving the implication that the Amiga is not selling as well as it should, could, or has to to meet expectations. My question to you is this: how many units should, could, or do you expect the Amiga to sell to be considered "selling"? My next question is: how did you arrive at this number and why?

For all its technological gimcrackery, the Amiga has some serious shortcomings. First of all, it's new, and that's bad. The vast, untapped masses of computer peasants have had an opportunity to test-drive VIC 20s and Commodore 64s. From the sounds of it, there aren't too many peasants left out there in the marketplace. The general level of computer awareness in the population has gone up considerably since the heyday of the Coleco Adam, TI99 and Radio Shack CoCo.

Secondly, the operating system is new, and worse, incompatible with the majority of the installed systems. It's the old 78 rpm vs 33 1/3 rpm syndrome: many people have too much invested to start from scratch again. So what that LPs have better sound reproduction?

Thirdly, the Amiga is not an open architecture system. It's amazing what you can do with an Apple or PC and the right add-on cards. Years from now, those systems will be upgraded to virtually match the latest technology. To upgrade the Amiga becomes an expensive kludge. Sure, the new Amiga 2000 may correct that, but your lament didn't cover the new machines, did it?

And finally, the Amiga is not cheap. Let's not compare it to IBM, though. So few people are buying real IBMs (mostly new clone manufacturers, I bet) that even IBM became concerned enough to try and market something different. What about the technofreaks? Most of them buy Ataris, which are much more reasonably priced, and then try to make them run like Amigas (Amigae?).

There *is* a niche for the Amiga, but it's a very small and specialized one, very similar to the niche that was carved out for a short time by the Ohio Scientific Challengers. The niche consists of the most eccentric technofreaks who also happen to have lots of discretionary spending power. Commodore created that niche, and emphasized it with their Andy Warhol/Blondie introduction to the world. Unless the price comes down, or the available software increases tremendously, the Amiga may die in that niche, just like the OSI C1P.

It's tantalizing to think what Jack Tramiel could have done with the Amiga if his brand of marketing had been followed. Instead, we end up with arguably the most powerful micro on the market today being widely seen as having no natural place. It reminds me of the Cord, which was too advanced for its time, or the Titanic, which succumbed to its own phantasy.

Are the consumers wrong for not buying it? Are the developers wrong for not churning out software? Or are our own expectations wrong? I suspect the latter.

By the way, I don't own an Amiga, simply because I can't afford one. My Commodore 64 will just have to do for a while longer. My OSI C1P is beyond resuscitation, and available to anyone with a good story.

John Kula, Victoria, British Columbia

Well, there's enough opinions in that letter to fuel half a dozen long conversations, John, but for now we'll have to make do with some quickie responses to the points you raise. Here we go, one by one. . .

Newness: it's hard to design a new computer that doesn't suffer from this "defect". The fact that the market is more aware nowadays than it once was should help the Amiga, not hurt it – one of the problems we see is that it's hard to appreciate its unique strengths until you've been around other computers for a while.

Operating system: yes, it's incompatible with everything, though Commodore has gone to a lot of trouble to make IBM compatibility available to those who want it. But there is no progress without change, even if it takes a while, and the switch from 78 to 33 1/3 didn't take all that long. Do you have a lot of 78s in your record collection?

Architecture: we just plain disagree on this one. Though there are some problems with the expansion interface on the Amiga 1000, they can be and are being overcome by dozens of hardware manufacturers. Commodore has been very forthcoming with the expansion specifications (one of the problems has been that the specs have continued to evolve after the release of the machine) and other data relating to the Amiga's internal operation; this is in contrast with the history of the Macintosh, which truly is a closed–architecture computer.

Cheap: okay, it's not that cheap. If you want some perspective, though, look at the early pricing of the Commodore 64, which came pretty close to the current pricing of the Amiga 1000. By the way, trying to make an Atari ST run like an Amiga is going to be a frustrating experience for anyone who tries it. . . sort of like trying to make your pop–up toaster run like a microwave oven.

Niche: We'll have to wait on a final answer for this one, but Amiga is already establishing itself well with artists, video professionals, musicians and engineers. Top–flight productivity software has been emerging for a while now, so we should start to see it making inroads into traditional business circles, especially with the recent release of the Amiga 2000. Whether the Amiga will make it as a "home computer" remains to be seen; we still believe that depends a lot on how well Commodore gets across the point about multitasking, which is for us maybe the biggest thing the machine has going for it.

As for your original question about numbers, it remains to be seen how far the Amiga can go. What's certain is that it has not been well marketed, and hence is not well understood by a lot of potential buyers. When that changes (and indications are that it may change soon), I guess we'll quit griping.

Mysterious quote mode explained: In a recent Transactor there was a Bit about how a value greater than 127 in location 646 ($0286) of the Commodore 64 affects the delete key in quote mode (Bits, Volume 8, Issue 1). This discovery intrigued me, and so I started looking around to see what really happens.

The delete routine contains this code:

```
e777 lda  $0286
e77a sta  ($f3),y
e77c bpl  $e7cb
```

. . .and it is followed by this code from the routines that handle the CTRL characters:

```
e77e ldx  $d4
e780 beq  $e785
e782 jmp  $e697
```

. . .we follow the jmp, and find:

```
e697 ora  #$80
```

. . .then a few bytes later

```
e6a2 jsr  $ea13
```

This last instruction is a call to the routine that prints the character currently in the accumulator.

It appears that the programmers assumed the value in 646 would always be less than 128, which should be the case considering the C64 only has 16 colours. Poking a value greater than 127 causes the branch at $e77c to fail, and the routine falls through into code that is intended to handle the printing of control characters. The delete occurs, and the cursor gets bumped back, then the character gets printed, which explains how the previous character gets corrupted. After printing the new character, the cursor is back where it started before the delete. The black hole effect is caused because the cursor gets moved back. Notice how the character printed reflects the value of 646.

Nickey MacDonald, Fredericton, New Brunswick

*Thanks for the detective work, Nickey. Now if only someone could come up with a **use** for this 'mode'. . .*

Guru mail department: *Jim Butterfield has passed on to us the following letter from reader Joel Rubin, who offers the following comments on Jim's article on secondary addresses and the Kernal (Secondary Address Bits, Volume 8, Issue 1):*

1) The Kernal OPEN routine ORA's the secondary address with #$60, at least on the C64 and C128 ($FEFD on the 128), so:

```
lda #2: ldx #8: ldy #$f: jsr setlfs
```

. . .is okay.

2) Since registers, unlike Basic file parameters, have no default value:

```
lda #3: ldx #4: jsr setlfs
```

. . .may well end up opening the printer the wrong way (e.g. if .y contained 7 going into the routine).

3) You do seem to need ORA #$60 if you use Kernal TALK, LISTEN et al.

Joel Rubin, San Francisco, California

To which Jim responds:

Right on all three counts. To give more detail:

*1) Earlier Commodore machines **did not** insert the extra secondary bits when you called OPEN ($FFC0). Thus, to set a secondary address 15, you had to LDY #$6F before your call to SETLFS; a value of #$0F wouldn't work. I write code for a wide variety of Commodore machines. . . and since it's as easy to "write in" those extra bits, I do so and save myself the trouble of needing to recode when I take it to another machine. On the newer machines (VIC 20 and subsequent), you can let the OPEN routine do the job for you. . . although it's no extra work to add the bits in your coding, and might help you remember that they are always a necessary part of the working SA.*

2) Agreed. As I suggest, if you don't want a secondary address you should still supply a value of 255 to signal "no SA".

3) Direct calls to TALK/LISTEN definitely need those bits, as you say; the system assumes that the SA was set up with the extra $60. Most users won't need to call TALK/LISTEN. . . they will do less work by using CHKIN ($FFC6, switch input path) or CHKOUT ($FFC9, switch output path); then GET ($FFE4, get a character) or

CHROUT ($FFD2, send a character); and finally CLRCHN ($FFCC, restore input and output default paths).

Jim Butterfield, Toronto, Ontario

More Plus/4 Tech Info: There is another source of technical information for reader Jim Welch, who was looking for a Plus/4 circuit diagram (Letters, Volume 8, Issue 1).

SAMS Computerfact has complete technical data and schematic diagrams available for the Plus/4. If they are not available in his area, they can be ordered from: Tenex Computer Express, P.O. Box 6578, South Bend, Indiana 46660. The product number is 33551, and the price is $17.95 (US) plus $3.75 shipping and handling.

If anyone has a machine language screen dump program for the Plus/4 that would work in monitor mode or in regular mode, I would like to obtain one. Thank you.

P. Wendt, Dodson, Louisiana

tRanzbloopeRs

The Blunderful Mr. Ed: If you typed in Chris Miller's nifty text editor, Mr. Ed, from Volume 8, Issue 2, may have been disconcerted to discover that the Verifizer code for line 5530 did not match the one printed in the magazine. The line in question is:

```
5530 jsr left ;dim
```

The Verifizer code given in the magazine was GE, but it should be OK. No, it shouldn't be *okay*, it should be "OK". How did the wrong Verifizer code get into the listing, you ask? We'd sure like to know.

Long Symass Labels: We don't know how many people have already noticed this, but it seems that all versions of Symass released so far will not properly handle labels over 8 characters long (what should happen is that longer labels will be accepted, but only the first 8 characters are significant). This will be fixed in the upcoming new version of Symass. . . meanwhile, be terse.

Space or Null String: Believe it or not, the typesetting equipment we use has no character equivalent to the quote (") seen so often in nearly every program listing. Until recently, the quote had to be simulated by rotating an inch mark (″) counter-clockwise 24 degrees. But for some reason, the typesetter rotate command crashes the preview screen that all articles are sent to before actually typesetting them. This was resolved by using yet another kludge. The typesetting computer allows variables just like CBM Basic. All quotes were substituted with a variable, and while previewing the variable was set to the character pair ''. Then, when the article was ready to be typeset, the variable would be re-set to the rotated inch mark. Often this final step was neglected which is why some listings appeared with quotes as '' instead of ".

When Attic acquired the COMPUTER font, we found another solution. This font has a quote symbol, but it looks like this: ". Yechh! However, in superscripted mode it looks great (") – and it doesn't crash the preview! Except for one last problem. When there's two side-by-side (ie. null string) it can appear to some as one space within quotes. Although it can usually be determined by deduction, we feel it may have been the source of some program entry trouble, especially since the Verifizer was designed to ignore spaces. Naturally we have a solution for this too; from now on the space between two quotes side-by-side will be "tightened up" so there can be no question that it means "null string". We hope that problems with quote marks (which would have been five years old this October) will be eliminated forever.

Now if we could just get CompuGraphics to create a decent looking uni-width font for program listings.

A few TransBasic bugs: *In Letters this issue, we quote from a letter by Nickey McDonald of Fredericton, New Brunswick, who gave us the result of his investigations into the "undocumented C64 editing mode" lately reported in Bits. Here we quote him again:*

I have typed every TransBasic module and seem to have them all working, but I found a few bugs along the way. First is some trivia, but I wondered if anyone else ever noticed: the heading "TransBasic Parts 1 to 8 Summary" never changed after instalment number 8. I miss TransBasic, and so started to make my own modules and even to convert other peoples' routines to TransBasic format.

Second is a problem with the TransBasic USE command. Line 7192 should read:

```
7192          jsr    errpgm
```

. . .and not "jsr errmem" as published. Line 7242 in the same module causes a problem if you specify a device number. This is because the line following "ldx device" is a three–byte operation. To fix this problem, it is a simple matter of changing two lines:

```
7242          jmp   uz4p5
7246 uz4p5    stx   t2
```

Third is the Labelled Goto module. The author did not store the destination line number when calling a line, so if an error should occur in the called line, the error handler reports the wrong line number. I made the following modification:

```
6036 lgot7 ldy #2: lda ($5f),y: sta $39
     : iny: lda ($5f),y: sta $3a: lda $5f
```

Capacitance Meter Line Numbers: *This one goes quite a while back, to the article The Commodore 64 Capacitance Meter, by Jim Barbarello of Englishtown, New Jersey (Transactor, Volume 7, Issue 4). Back in December Jim sent us a correction, which we promptly misplaced. . . only to have it turn up again the other day. Here's an extract from Jim's letter, with our apologies for the delay:*

Some letters I have received indicate a problem following the "Optimizing Performance" instructions (page 32). It seems that modifying line 110 has no effect.

Upon review of the article, I noticed you renumbered the program, but did not change the line reference in "Optimizing Performance"! Please print a correction indicating that the two references to line 110 should have been to line 200.

First–Aid for Programmer's Aid: *From the same pile of correspondence that yielded Jim Barbarello's correction above comes this from James G. Rae of Chillicothe, Ohio. Referring to the article*

Programmer's Aid for the Commodore 128 by Joseph Caffrey (Transactor, Volume 7 Issue 5), James writes:

The program has three known problems. The first, reverse printing of the first character on the line in the 40 column mode only looks bad and causes no real problems.

The second occurs when a program line uses more than one screen line and one of the additional screen lines starts with a number (SYS, GOTO, etc) that matches an existing line number. The scrolling routine will read this as the last line number listed and continue listing from the next line number. This is a major problem in the 40 column mode, but can be lived with in the 80 column mode by limiting all lines to one screen line.

The third problem is a computer lockup when scrolling up past the first line in the program if there is anything appended to the program between the end of the program and the end of basic pointer. When this occurs, the technique of assuming the end of program link is at the end of Basic minus two is incorrect, and causes the "rstlink;" routine to go into an endless loop.

I have rewritten the "foundup" routine to eliminate this problem by executing it twice at the first to last line transition. The first time it looks for the pointer pointing to the last line link, while the second time it looks for the link pointing to this link.

The revised assembler code below is 9 bytes shorter than the original, so the revised code can simply be inserted in the original.

Note: the following Basic program line will set AD as the address of the byte following the last line link. This can be compared with the end of Basic to determine if any bytes have been appended to the program:

```
63999 ad=peek(4610)+256*peek*4611)+35: return
```

Here is my version of the "foundup" routine:

```
foundup jsr   putback
        jsr   huntline
        bcc   upout
        jsr   scrldwn
        ldx   linkpntr     ;Moved next stx to start of init
        lda   linkpntr+1   ;Moved next sta to start of init
        cmp   sob+1        ;Branch if last line listed was
        bne   find         ;not the first line in the program
        cpx   sob
        bne   find
        jsr   scroldwn     ;Insert two blank lines
        jsr   scroldwn
        lda   #0           ;Set Oldlink at $0000
        tax
        jsr   init         ;Find the address of link
        ldx   linkpntr     ;pointer to $0000 and
        lda   linkpntr+1   ;put into Oldlink
find    jsr   init         ;Find address of link pointing
        jmp   setup1       ;to Oldlink and then jmp to the
                           ;line Lister routine
                           ;
init    stx   oldlink      ;Relocated stx and sta opcodes
        sta   oldlink+1
        ldx   sob          ;Start search at Start of Basic
        lda   sob+1
rstlink stx   linkpntr
        sta   linkpntr+1
        jsr   rdnwlnk      ;Read value of next link (x,a)
        cmp   oldlink+1
        bne   rstlink      ;Loop until linkpntr points
        cpx   oldlink      ;to Oldlink
        rts
```

Xref64: I'm afraid there are two bugs in Xref64, but thankfully both are pretty easy to fix. So here goes.

Change the third number on line 1530 from 217 to **214**.
Change the last number on line 1770 from 165 to **76**.
Change the first two numbers on line 1780 from 30 and 208 to **198** and **166**.

Replace lines 3270-3310 with the following lines:

```
3270 data  82,  65,  77,  13,  13,   0, 165,  29, 201
3280 data   6, 144,   4, 169,   5, 133,  29, 165,  30
3290 data 208,   3,  76, 134, 161,  76, 145, 161
3300 data   0,   0,   0,   0,   0,   0,   0,   0,   0,   0
3310 data   0,   0,   0,   0,   0,   0, 162, 6.1, 160
```

I'm sorry for any inconvenience these bugs may have caused. (I hate bugs!)

Sincerely, David Archibald

Editor's Note: *If there's space, and there should be, we'll put new versions of the C128 Programmer's Aid and Xref64 on Transactor Disk #20 for this issue.*

Division Revision: *For the following correction we are indebted to Ross Churchman of Thunder Bay, Ontario, who writes:*

In the second part of the article "High Speed Integer Multiplies and Divides" (Transactor, Volume 8, Issue 1), there are several typing errors that can be misleading if not noticed, as was the case the first time I read the article. All occur in the section on Division (page 43, second column). The line "M1/M2 = R1 rem R2" looks as if M1 is divided by M2, whereas M1 is actually dividing into M2. This should have appeared as:

$$M1\ )\overline{M2} = R1 \text{ rem } R2$$

In Step 5 at the bottom of the same column, the line "CMP R2–M1 (carry clear)" should have read "CMP R2–M1 (carry set)". At this step, R2 is equal to M1 in the example used, and subtracting with the carry set, as the ML code does, will leave the carry still set.

There is one further mistake, in the ML coding of this division algorithm (page 45, second column). The fourth line, which reads "ROL M2+1" should have read "ROL R2+1". This makes the code compatible with the earlier steps and description.

As a computer hobbyist and teacher of community night courses in machine language and computer interfacing, I enjoy your magazine very much. Keep up the good work!

TeleColumn

PunterNet **By Geoffrey Welch, Toronto, Ontario**

Most Transactor readers will be familiar with the name Steve Punter. Since he wrote his first BBS (Bulletin Board System) program years ago, 'the Punter BBS program' has gained recognition as the most polished BBS program available for the Commodore 64. However, the one thing that Commodore-based BBSes lacked that was available for larger computers was the ability to 'network', to have messages transferred from one BBS to another. Although networking had been discussed several times before and methods of doing it had been thought of, no one had actually produced a working network. Steve Punter, seeing that the interest in networking had left the stages of theoretical discussion and was nearing the experimentation stage, worked out his own solution and put it into testing with six boards in the Toronto area starting in the fall of 1985. The system proved reliable and, in March of 1986, was made available to the general public. I was the first to sign up and have been operating 'Node 7' ever since.

From the beginning, the system has been 'a hit'. As I type this, 89 boards have registered to join the network – about a third the number of BBS programs Punter has sold – and about 60 are active. PunterNet users can send mail from Halifax in the east to Hawaii in the west, south to Miami, and off-shore to Bermuda, in some cases for less than the cost of a stamp! Many of the active boards are in the Toronto area, and Toronto users take advantage of this to participate actively on several boards while only having to sign on to one.

The BBS program itself didn't look different. The first thing an unsuspecting user saw were messages from people whose first names started with a number and a slash. "1/Steve Punter" and "7/Geoffrey Welsh" didn't exactly look like names a loving parent would give their children. . . those inspired enough to read bulletins knew, though, that this was simply how the BBS indicated that the message had come from another board. The number before the slash told you which 'node' the message came from so that, when you replied to the message, the BBS could make sure the message was routed through the correct board and arrive in the recipient's mailbox.

For instance, if Steve Punter wanted to send me a message on my board, he would tell his board to send the message to "7/Geoffrey Welsh" and, after midnight that night, his board would call node 7 (my board). When it got through, it transferred the message and the message would appear on my board as having come from "1/Steve Punter". Since our boards are within local (free) calling distance, the board will deliver the message the same night. Obviously this could prove expensive if the destination was long-distance, so the program offers the message sender two options for long-distance messages: regular and fast delivery. Regular delivery keeps messages going to a given destination in a file until there are four messages going to the same place (so the cost of the call is distributed over the four messages) or until the message is four days old (to make sure that the message doesn't sit and wait forever before it's sent). If the message is important and must go through immediately, the sender can designate the message for fast delivery, which means that the board will not wait for the four messages or four days, but send it immediately. For this luxury, the sender pays double the normal cost of sending the message.

That, of course, brings up the cost of sending the message to begin with. If your node is within local dialing distance of another, it doesn't cost anything to send a message to a user there, so SYSOPs (SYStem OPerators) usually let users send messages to local nodes free. But contacting most boards requires a long-distance call, and that costs money. Fortunately, the network calls are all made while the telephone company offers the highest discounts. Still, the call costs money and the SYSOP must pass on that cost to his/her users, and the BBS program has a built-in accounting system to handle this. Only users with money in their account may send messages to long distance destinations, and the program automatically calculates the cost of a message (based on the length of the message and a minimum charge) and deducts that from the user's account. The exact charges are completely up to the SYSOP, so they vary from board to board.

Further savings are possible using 'pathways'. For instance, there are over a dozen network nodes in the Toronto area. Why should a California (or, for that matter, Hawaii) node call each of these separately, when they are all a free call away from each other? The pathway concept allows SYSOPs to route messages to nodes other than their actual destinations, so that all messages to the Toronto area can be sent in a phone call to one node, and the Toronto node would then distribute the messages to their destinations. Although this is most effective in Toronto, where a single call can distribute messages to nearly twenty boards, there are other areas where two or three nodes can call each other free of charge. Naturally, only boards with storage space to spare can handle the heavier loads that result from using them as pathways, but the concept works well.

But how does any node (and its users) know what other nodes are on the network? This is accomplished by two files – "NetUpdate" and "NODES". The first is a file containing the latest list of nodes and the information about them: their telephone number, their speed (300 or 1200 baud), if they are temporarily down, etc. The NetUpdate file is kept up to date by Steve Punter at PunterNet 'headquarters' here in Toronto and is automatically fetched by any board that sends a message to node 1 (or, if the caller is using a pathway to get to node 1, whenever it passes a message to the board it designated as the pathway to node 1). The second file, NODES, is just a bulletin that the board fetches along with the NetUpdate file IF the SYSOP has told it to do so. Users can see the NODES file by entering the bulletin section and typing NODES.

Thanks to PunterNet, I keep in touch with a friend in Kitchener, Ontario, and people in Los Angeles, Milwaukee, Montreal, Allentown, Vancouver. . . people I would never had heard of if it weren't for PunterNet. And it only costs me a couple of dollars a month!

A Switchable RS–232 Interface

Mark Farris
Tulsa, Oklahoma

A do-it-yourself "VIC1011A adapter" for true RS-232 at the user port!

I recently had a chance to purchase a surplus 1200 baud Hayes compatible modem. These external modem cards were complete except for the power supply and, of course, the interface to connect it to the 64's user port. Building the interface is what made the difference between a "good deal" and paying full retail to upgrade from 300 baud. I was fortunate enough to have some of the necessary parts on hand, such as the user port connector and the capacitors, resistors and diodes. This saved about half the cost of buying all the parts for the project.

Preliminaries

In theory, adding an RS 232 port to the 64 would allow the connection of just about any device that provides an RS 232 connector. In practice, there are software and hardware differences that make this type of universal connection difficult. The following interface was designed to work with a stand alone Hayes compatible modem, but as long as the software requirements are understood, it should work with just about any DCE device. (See Martin Goebel's "Universal RS–232 Cable" Vol 7, issue 4, for an explanation of DTE vs. DCE devices).

The lines to and from the modem can be separated into 2 groups: Data lines and Control lines. Data lines carry the actual data to and from the modem, and Control lines control that data flow. Another way to think of it is: data that pass via the Data lines go through the modem, down the phone line, to the destination computer and back again. There are 2 Data lines: Transmit Data (TxD) is an output from the computer and Receive Data (RxD) is an input to it.

Control lines, on the other hand, go to and from the modem itself. These lines handshake between the modem and the computer, and allow the software to make decisions based on the presence of high or low signals at the user port. The Commodore 64 provides 7 control lines, 6 of which are standard RS–232 control lines: Request To Send (RTS), Data Terminal Ready (DTR), Ring Indicate (RI), Carrier Detect (DCD), Clear To Send (CTS), and Data Set Ready (DSR). Most modems provide all 6 of these control lines, but some modems also provide a Speed Indicate (SI) line. Commodore didn't provide a SI line on the 64, but fortunately they did add an extra unassigned pin on the user port. This unassigned pin (J) can be used as the SI handshake, as well as any other control line. If pin J on the user port is to be used as a Speed Indicate line, the software would be responsible for that configuration. This brings up the question of software compatibility.

User Port RS–232 Configuration
* = Control line

Pin	Assignment	
A,N	Ground	
B,C	Receive Data	(RxD)
D	Request To Send	(RTS)*
E	Data Terminal Ready	(DTR)*
F	Ring Indicate	(RI)*
H	Carrier Detect	(DCD)*
J	Unassigned	*
K	Clear To Send	(CTS)*
L	Data Set Ready	(DSR)*
M	Transmit Data	(TxD)

Compatibility

For some reason, not all commercial interfaces connect every control line available at the user port. One popular interface only connects 4 of the 7 control lines: CTS, RTS, DTR, and DSR. Those 4 lines are switchable, which makes for a more versatile interface, but the fact that the DCD, RI, and SI lines are left out makes that interface incompatible with many types of software. Some Bulletin Board software "looks" at the Ring Indicate line for an incoming call. The Commodore VIC 1011A interface does not connect the RI line, making it incompatible with some of the most popular BBS software written for the 64. There is no standard method for accessing the control lines through software. Programmers can, and do, use different lines to accomplish the same thing. The DSR pin tells the computer when the modem is turned on, and the DCD indicates the presence of a carrier. But the DCD line could be used in the place of both of these lines. After all, if there is carrier present, it goes without saying that the modem has power. There are also those programs that are written under the assumption that a Commodore modem will be used. Programs written for use with a 1670 will not always work properly with a Hayes compatible modem.

A Better Way

The problem of compatibility can be solved by building an interface capable of adapting to different software requirements. This is accomplished by connecting all control lines and making each one switchable. Digital signals are expressed as either being "high" or "low", with a high being in the +3 to +5 volt range and a low being close to 0 volts or Ground. The user port works on this (TTL) digital principle. The RS–232 standard as set by the EIA does not follow this TTL system. RS–232 devices can

send a wide range of voltages to and from each other. They are not limited to 5 volts, and quite often use a 12 volt level if there is some distance between them. Instead of using positive voltage as a "high", RS-232 assigns a negative voltage as the high signal.

The RS-232 standard dictates that signals between devices will be + or - 15 volts with the positive voltage being a "low". Input and output voltage on this interface will be in the +/- 5 range.

Negative voltage is considered a high in any RS-232 cable, but applying a negative voltage to the 64's user port can damage the computer. So, it's the interface's job to convert the RS-232 signals to signals usable by the user port. Inputs to the user port from the modem's RS-232 connector must first go through a MC1489 Quad Line Receiver IC. It changes the -5 volt (high) and +5 volt (low) signals to normal TTL levels. Outputs from the user port must go through a MC1488 Quad Line Driver IC. It takes the user port signals and converts them to RS-232 type signals.

Control lines are not treated the same as Data lines by the 64. Data lines are configured as "active low" on both the modem and computer. That is, the line is ON when it is at 0 volts (Ground level).

Control lines are "active high" at the user port (line is ON when +5 volts is applied). Since the modem wants all lines to be "active low", the Control lines must go through an extra inversion. Most commercial interface designs use a 740 4 Hex Inverter IC to take care of this inversion, but there is a better way. To make the Control lines switchable, they are connected through 2 74LS86 Quad XOR Gate IC's. Each Control line is connected to one of the 2 inputs on each gate. The other gate input is hooked through a DIP switch to ground. A 10K resistor is connected to +5 volts in parallel with the switch so that the DIP switch selects either +5 volts or ground. The truth table for the XOR Gate IC shows that only an odd number of inputs produces a high output. This means that when the switch is open, there will be a high on one input. Any signal coming in on the control line will be inverted. Closing the switch lets the signal go through uninverted. The normal setting will be all switches open but, to satisfy different software requirements, each line can be changed independently.

Construction Tips

Radio Shack's Digital IC experimenter's circuit board is ideal for this project. The user port connector solders directly to the "plug in" edge of the circuit board. It is not necessary to trim the circuit board to make it the same width as the user port connector. There are 2 separate supply busses on the circuit board. Connect Pin 2 on the user port to one of the supply busses. This will be the +5 volt (Vcc) supply to the ICs. Connect Pins 1, 12, A, and N to the other supply buss. This makes the Vcc and Ground supplies easily accessible from anywhere on the board. It is then a simple matter of placing the IC's on the board with the supply busses running directly under each chip. Solder pins (14) and (7) on the IC's to the proper buss (except U1).

All IC's get their supply voltage from the +5 buss except the 1488 Quad Line Driver. The 1488 requires a Vcc of at least +7 VDC and a Vee of at least -2.5 VDC. This supply is taken from Pin 11 on the user port. Positive going cycles of the 9 VAC go through D1, and are filtered by C1. This supplies pin 14 of the 1488 with Vcc. Negative cycles go through D2, are filtered by C2 and supply pin 7 with Vee. C3 and D3 serve the purpose of equalizing both the positive and negative voltages from pin 11 on the user port. Without the addition of the extra Diode and capacitor, the voltage on pin 14 of the 1488 IC would measure close to +12 VDC and the voltage on pin 7 would be approximately -1 VDC. With the addition of the equalizing circuit, both sides should measure between +/- 10 to 12 VDC. This is well within the maximum of +/- 15 VDC allowed.

After connecting the supply circuit to U1, plug the circuit board into the user port and measure the voltage on pins 14 and 1 to verify that pin 14 is getting approximately +10 volts and pin 1 has -10 volts. (Make sure the computer is OFF while plugging it in). Next, place the DIP switch on the board with one row of pins in the Ground buss. Connect the other end of each individual switch to the proper pins on the 74L S86. Now the 10K resistors can be connected from the +5 volt buss to the pins on the 74LS86 IC. Connect one of the ribbon cable wires to the correct pin on the DB25 connector. Go ahead and connect the wire all the way from the DB25 connector to the user port, through the 74LS86 when necessary, thus finishing each line one at a time.

Go slow and follow the schematic. A different colored wire for each line would make tracing the circuit easier later on, so rainbow colored ribbon cable would come in handy here.

After all Control and Data lines are hooked up, solder the .1 mf capacitors from pin 14 to the ground buss on all IC's and from pin 1 to the ground buss on the 1488. Solder the 470 pf capacitors from pins 6, 8, and 11 on the 1488 to the ground buss. Now solder all unused inputs on the 1488 and 1489 to the ground buss. This should complete construction of the interface.

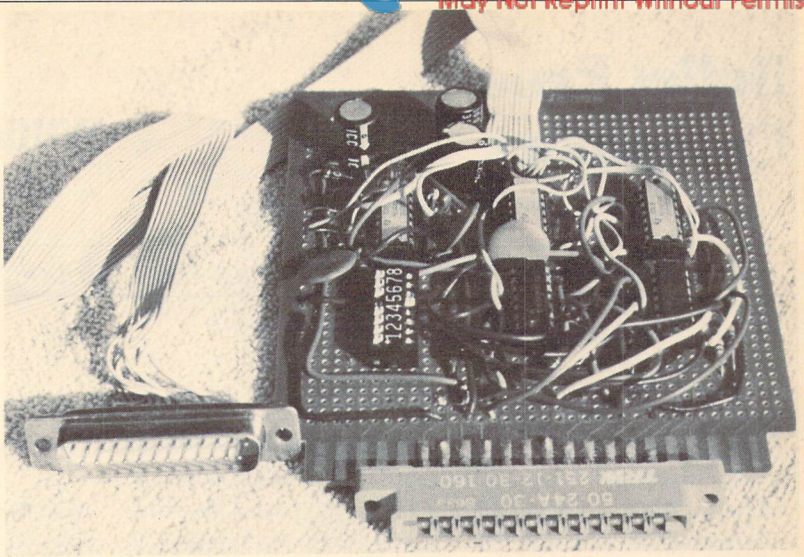
If everything is hooked up right, there should be 2 unused gates on U2, 1 unused gate on U5, and 1 unused switch on the 8 switch DIP. The schematic shows one line out of U2 going to the unused gate on the 74LS86 hooked to the unused switch. This line can be kept as a spare if, for some reason, one of the other lines develops a problem.

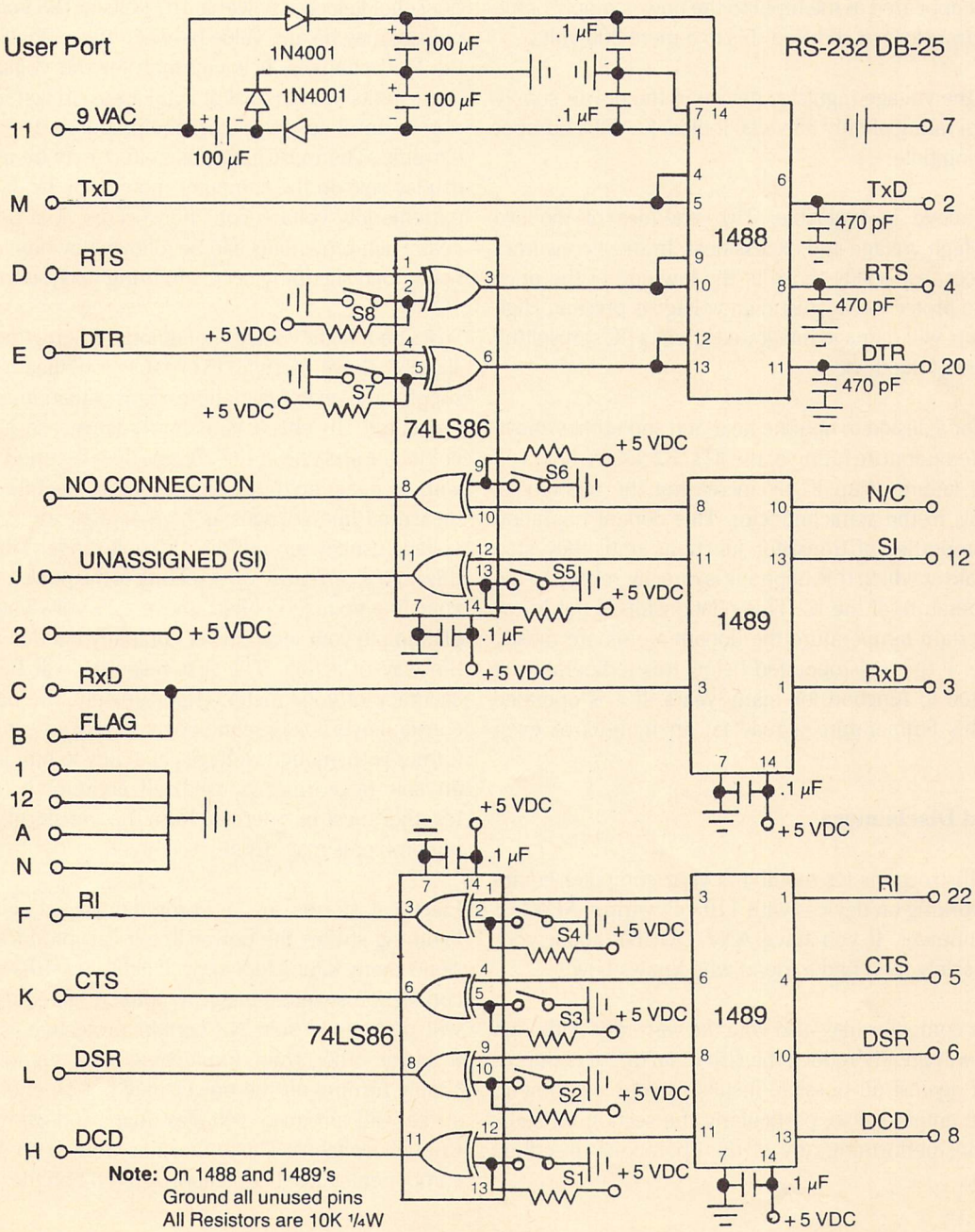
Final Thoughts

Radio Shack part numbers are listed for some components. Just about any electronic parts supplier should have most parts stock including the 74LS86 XOR Gates. Ordering all parts by mail would probably be the least expensive way to go. Use only LS versions of the 7486 XOR gate IC. These use very little current and there is only an extra 100 mA available at the user port. If there is a software compatibility problem, experiment with the switch settings. Closing switches to the DCD, DTR, and CTS lines will emulate a 1670 modem with some software.

Parts List

1	Experimenters circuit board (Digital IC) 276–154A
1	12/24 .156 edge card connector (user port)
1	DB–25 connector (25 pin RS–232) 276–1547
3	100 mf 35 volt electrolytic capacitors
3	470 pf disk capacitors
6	.1 mf disk capacitors
8	10K 1/4 watt resistors
3	1N001 rectifier diodes
1	1488 Quad Line Driver IC 276–2520
2	1489 Quad Line Receiver ICs 276–2521
2	74LS86 2 input Quad XOR Gate ICs
1	Miniature 8 rocker DIP switch 275–1301
	18-36 in. of ribbon cable (11 or more conductors)
	Several feet of 22 ga. or smaller hook up wire





Note: On 1488 and 1489's
Ground all unused pins
All Resistors are 10K 1/4W

Bullet Proof Computers

Evan Williams
Williams Lake, British Columbia

. . .two enemies of modern electronics are high voltage and excess heat. . .

It was a dark and stormy night. The clock struck twelve. With a brilliant flash and a clap of thunder, a 50 megavolt bolt of lightning struck the high ground wire on the power line three kilometers from the house. This induced a 2000 volt transient. In microseconds it appeared at the input to the power supply of the computer. The transformer reduced this to a mere 200 volts.

Unfortunately, the voltage regulator device in the power supply had a maximum rating of only 35 volts. It died. So did a number of parts in the computer. . .

The scenario above is avoidable. Two enemies of modern electronics are high voltage and excess heat. In most consumer electronic devices, especially those at the low end of the price scale, little or no protection against high voltage is present. High voltage transients will damage integrated circuits (IC's) resulting in prompt failure of the devices.

It is also usual for a design to operate near and sometimes above the maximum temperature rating of the IC's. Excessive heat will cause eventual failure of an IC by increasing the mobility of dopant materials in the semiconductor. This dopant migration changes the properties of transistor junctions until they stop working. The rate at which this happens is directly related to the operating temperature of the IC. This relationship is not linear and below a certain temperature the dopant atoms are mostly locked in place. If the IC is operated below this temperature it may be expected to function for many years. If it is operated much above this temperature it may fail in months or even weeks.

Warnings and Disclaimers

The following instructions for modifying your computer equipment include working on devices with 110 volt wiring. ALWAYS unplug all equipment. If you have ANY DOUBT about your ability to work safely then find a friend who knows how.

Modifying your computer may also void the warranty. Although these changes will greatly reduce the risk of failure, it is impossible to protect against all possible insults to your equipment. Please read the entire article, particularly the section on static electricity, before performing any of the operations described here.

The Power Source

In North America, the standard line voltage supply is commonly referred to as 110 volt AC. In fact, the normal specification for household electric power is 107 volts to 125 volts AC. The actual voltage may be any value between these limits and will vary as the load changes. It is common for the voltage to go beyond these limits in industrial or rural areas. In general, low voltage is not harmful to computer equipment unless the situation is extreme. The most noticeable effect may be a reduction in the display size on the computer monitor or TV. If a "brownout" or extreme low voltage condition occurs, it is best to turn off all gear. Such brownouts can be followed by high voltage surges as major load shedding occurs to bring up system voltage.

Sustained high voltage conditions where the voltage is only slightly above normal can cause overheating of the power supply and eventual failure. This situation cannot be easily prevented. To check your line voltage use a good quality ac voltmeter and check the voltage over a period of days. A meter with a peak hold function is most useful. The author has measured line voltages as high as a steady 130 volts with the voltage usually around the 125 volt range. When the voltage is this high it can cause overheating of the power supply. If you find your line voltage is often above 125 volts you might try complaining to your local power company but don't expect much in the way of action. The first response will be to question the accuracy of your meter. Unfortunately, the only remedy for a constant overvoltage condition is to buy an expensive autotransformer with enough wattage capability to handle all your gear. If an autotransformer is used, all equipment that is interfaced together must be operated from the transformer. If not, grounding problems may result.

Electrical storms are a major danger in many areas. When lightning strikes the power line it can produce damaging transients many kilometers away. If lightning strikes near the service connection severe damage to some or all equipment connected will probably result. No certain protection is possible in this instance other than unplugging ALL equipment. Note that simply turning off the power may not be enough since a close strike will produce voltages high enough to jump straight through switches. The author has worked on equipment where components have been vaporized right off the circuit board by a

nearby strike. If an electrical storm is overhead or nearby, turn off and unplug your equipment.

The more common situation is when lightning strikes the power line somewhere in the same county. This may produce much smaller but still damaging transients. These can be handled by a device called a metal oxide varistor (MOV). The MOV presents a resistance to electrical current which is voltage dependent. When voltage across the MOV rises to the rated value, the resistance drops to a very low level, in effect becoming a short circuit. This prevents the voltage rising much higher and is called "clamping". MOV's can withstand very high currents for short periods of time. By placing a MOV across the hot and neutral wires of the power line, protection against transients is obtained. The power distribution system in the home should not be modified so the way to do this is to install a MOV in a power bar. If you are using a power bar that indicates it has surge protection then it already has such a device. Do not confuse this with the required circuit breaker found on all power bars. The circuit breaker is there to protect the power bar from overload, not the equipment plugged in to it.

If you have a power bar that can be disassembled, it is very simple to install a MOV.

Modifying a Power Bar

Unplug the power bar and all equipment. Remove the power bar back cover. If the power bar uses ordinary receptacles it will most likely be wired using push-in connectors on each receptacle. The end receptacle should have two free push-in connector positions (see figure 1). Bend the wires of the MOV (Radio Shack #276-570 or #276-568) to the correct spacing with right angle legs so that when pressed in to the connector holes the MOV will lie flat against the receptacle body. Install the MOV by pressing the wires into the holes. One wire should go into a hole on the hot side and one on the neutral side. It doesn't matter which wire goes where since a MOV is non-polar.

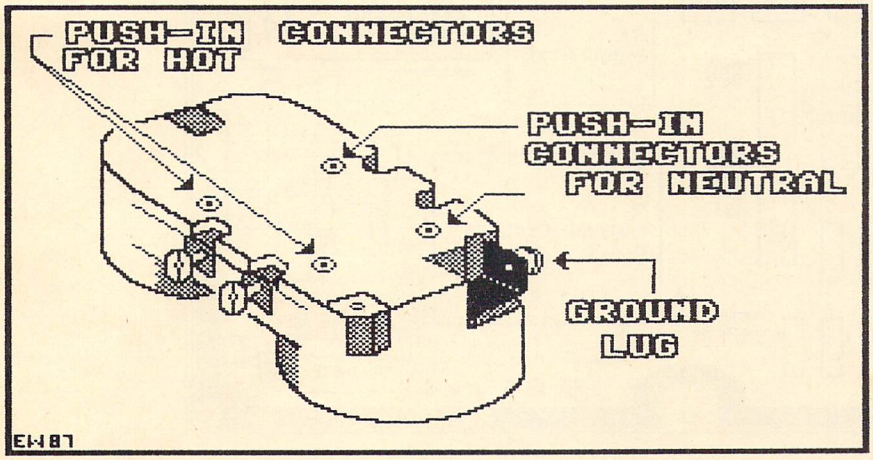


FIGURE 1

For complete protection against common mode transients as well as differential mode two more MOV's may be installed, one from the hot side to the ground lug and one from the neutral side to the ground lug. Tug lightly on the wires in the connector holes to ensure they are properly trapped. Tape the MOV to the receptacle with electrical tape. Reinstall the back cover of the power bar and you are done.

Additional Considerations

If you live in an area with underground power service do not assume your system is safe. The power distribution system always has some overhead connections that can be struck by lightning (ie. street lamps and traffic control devices).

If your computer is connected to a television for display purposes this presents another path for destructive energy, especially if the television uses an outside antenna. Yet another path exists if a modem is connected to the phone line and the computer.

Again, the only sure protection during an electrical storm is to unplug all devices that could provide a path for destructive energy to your computer.

The Computer Power Supply

Many home computers use an external power supply. This is done for a variety of reasons. The most important reason is that using a separate supply limits the 110 volt ac power distribution to that area only. This makes testing and approval by organizations like U.L. and C.S.A. quicker and easier. It also removes a major source of heat and bulk from the main computer case and allows more attractive design. Unfortunately, moving heat to another box does not eliminate it.

Some Commodore power supplies are notorious for failing early, especially the C-64 units. It is essential that the power supply be placed in a cool, well ventilated area with no obstruction of air flow. It helps if the unit is raised by adding small feet to it or placing it on blocks. It should never be placed on a carpet or in a closed compartment. The best solution is to place a small quiet fan so it moves air over the unit. If you have an early model C-64 power supply it may be of the repairable type. The author has such a unit and has drilled numerous 1/4 inch holes all over the case (after removing the internals!). This helps tremendously in keeping the unit cool. All later model C-64 supplies are NOT (easily) repairable as the components are potted in epoxy making the unit a solid brick. This does not help the cooling. The VIC-20 supply runs cooler as the power demands are

less than the C-64. The C-128 power supply runs much cooler (better design) and should not present a problem.

If a problem appears with the C-64 supply it is advisable to replace it with a second source unit such as the CPS-10 made by Estes and available from Jameco Electronics for US $39.95. This unit includes 110 volt ac outlets with spike protection. Also available from a number of sources are fan/outlet units with spike protection in the $30 to $40 price range.

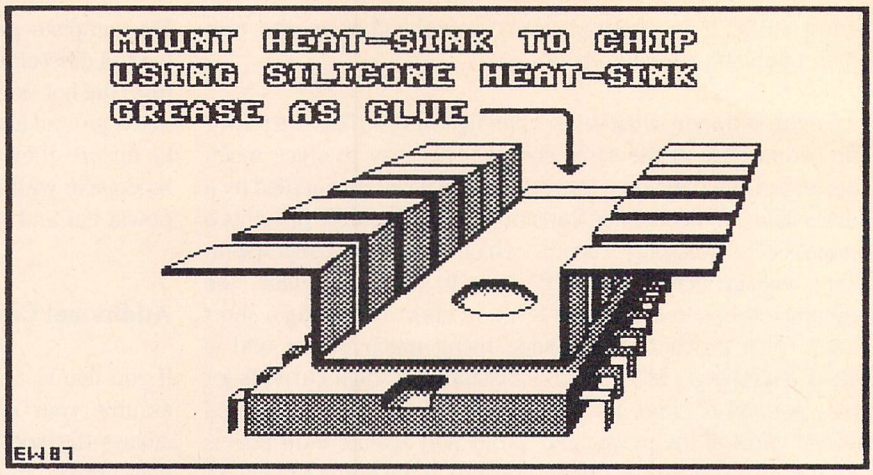


FIGURE 3

Heat and the Computer

From the UNIVAC to the C-64 heat has been a problem. The ROM's in the authors PET 2001 ran so hot they could not be touched comfortably. This is a sure sign that an IC is too hot. A thermocouple measurement showed a temperature of 71 celsius (160 F). The maximum safe operating temperature for most IC's is 70 degrees celsius (158 F). Some IC's in the C-64 and 1541 disk drive also run hot. The solution is to install heat-sinks on the IC's. The most suitable type are heat-sinks designed for TO-220 semiconductor devices. These are available at most electronics stores including Radio Shack (catalog number 276-1363).

Open the case of the computer or disk drive by removing the screws found in the underside. Carefully lift the top half of the case off. Most C-64's will have a foil covered piece of cardboard over the circuit board. This is a RF shield. It is also a heat trap. In most cases the author has found little or no increase in TV or radio interference if this shield is removed. Early C-64's did not have this shield. It is not illegal to remove this shield but if your equipment causes interference and a complaint is made to the Department of Communications you will have to correct the problem. Removing the shield can degrade the picture if you are using a TV set as your monitor. Experiment by tearing the small copper tape at the rear and bending the entire shield towards the front of the computer. Place the keyboard loosely on top of the lower case with the shield sticking out the front and turn on the power. If no noticeable interference exists on your TV or radio then you are safe in ripping the shield out. Removing this shield will not harm your computer, but will void the warranty.

Next, leave the computer running for half an hour. There are no dangerous voltages present in the computer. Watch out for rings, metal shirt buttons, necklaces, chains or any other metal objects that might fall in the machine and short out power. Although not dangerous to the user it could fry your computer. After it is warmed up, open the lid and place your finger on each large integrated circuit. Several IC's will be found that are quite warm. Also, the video circuits metal box (see figure 2) will be warm. To attach heat sinks to the IC's and the video circuits box the author uses only silicone heat sink grease (Radio Shack part number

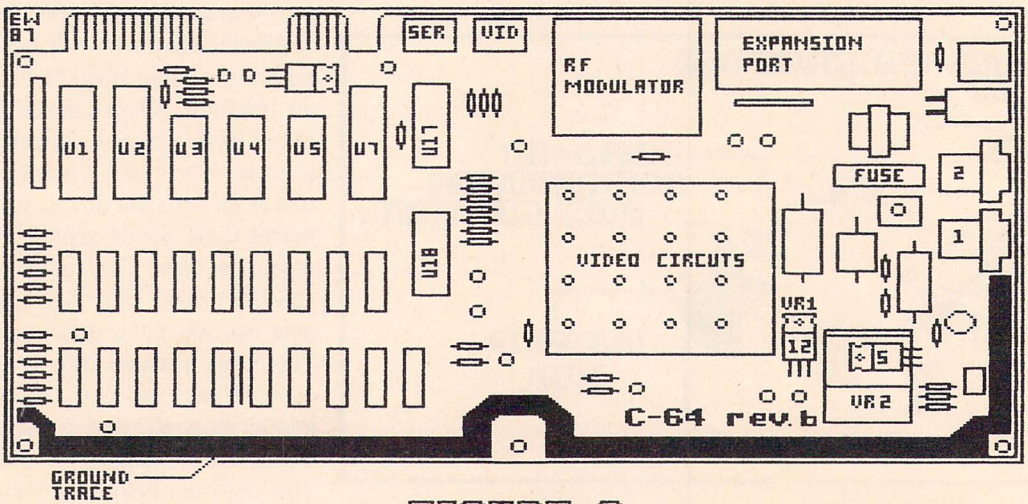


FIGURE 2

276–1372). This is sticky enough if the computer is not moved much. Use just enough to cover an area on the IC equal to the size of the heat sink. Place the heat sink on the IC and press into place (see figure 3). If the computer is moved around a lot use just enough grease to cover two thirds of the heat sink area on the IC. Cover the other third with contact cement and when tacky press the heat sink into place. The same procedure is used on the video box. The author used three heat sinks in a row for the video box. This overall procedure should also be carried out on the 1541 disk drive. Be careful to check for clearance of the heat sink fins with surrounding parts and the upper case. If interference exists the heat sink can be hacked, bent or chopped as needed.

The C–128 has an all metal shield covering the entire circuit board closely. In this shield are cutout tabs that contact certain IC's below. On two of two C–128's examined the tabs were not contacting properly and did not have enough heat sink grease. The C–128 shield is fastened in place with a series of screws around the perimeter and some small bent metal tabs from the lower shield. By removing the screws and straightening the tabs the shield may be lifted off. The heat sink tabs should be reformed to ensure good contact with the IC's and a generous amount of heat sink grease applied. Reinstall the shield. Do not overtighten the screws holding it. Rebend the small metal tabs which provide contact with the lower shield and reinstall the case.

- Never cover your computer gear with anything while it is operating; no magazines, no disk jackets or anything else.

- Always leave some space for air circulation around the equipment. Do not stack disk drives unless a fan is used to cool them.

- Operating the equipment in the direct sun is a good way to overheat it. Remember, the primary cause of component aging and eventual failure is excess heat.

All of the author's computers and related accessories have been overhauled as above. Some of the systems operate 24 hours a day. Since 1978 only two component failures have occurred and one was an early failure in the first month of ownership. The second was induced by zapping the joystick port on a C–64 with static electricity.

Static Electricity

Static electricity is a charge caused by friction. Walking across a plastic carpet on a dry winter day can build an amazing charge on your body. Touching any conductive object at a different potential can cause the charge to arc over. The insulating value of dry air is about 20,000 volts per centimeter. Anyone who lives in a dry climate has probably seen a spark at least this long jumping from their finger to another object. The maximum voltage that most IC's can withstand is about 25 volts or less. A static charge large enough to damage an IC can be so small that no evidence of it exists to the senses. Certain parts of all computers are easily damaged by the slightest touch when conditions are just right (wrong!).

The C–64 has two very vulnerable joystick ports right next to the often used on/off switch. These ports use exposed male pins on the computer side (arrrggghhh) so as to be compatible with Atari joysticks. Under no circumstances should these pins be touched. Worse yet, on the C–128 the reset switch is right next to the joystick ports.

The solution is to remove static charges from the user. The simplest way is to touch something metal that is grounded to the same outlet as the computer before touching any part of the computer. This is not always practical and there is a better way. An expensive solution is to buy an anti–static mat to place the computer on. This is connected to the outlet ground with a wire and drains static charges painlessly when touched. The author uses a cheap but effective method. Buy a small roll of 3/16 inch tin plated copper tape from a stained glass supplier. Stick strips of this tape to the side of the computer surrounding the joystick ports so that anyone plugging in a joystick cannot avoid touching the tape. Bring one end of the tape inside the case. Solder a one megohm resistor to the tape and with a piece of flexible wire solder the other end of the resistor to the ground trace on the main circuit board (see fig 2 for C–64). On the C–128, solder to the metal RF shield. Wrap the resistor with electrical tape or shrink tubing and close the case. The resistor is necessary to prevent a sudden surge of voltage. This modification costs very little and is effective with small children.

When doing any work on electronic equipment is is advisable to wear cotton clothing that does not produce static charges. Always make sure that you touch something grounded before touching any components. Before soldering, touch the tip of the iron to ground or use a grounded iron. It is best to avoid disassembly at all if the air is really dry and static is zapping everywhere.

A Final Observation

The author has always followed the practice of "burning in" new electronic equipment. This means just setting the new unit up, turning it on and leaving it on 24 hours a day for at least a week. For some reason, this seems to prevent future failures. The author has no certain explanation for this but there are some effects such as "oxide healing" that could explain it. It seems best not to subject the new gear to on/off power surges until this burn–in is done. The author is a confirmed techno junkie and buys every new widget that comes along. All items are burned in. Over the years almost no failures have occurred from clocks to stereos to computers. This is a GOOD THING. The author repairs computerized devices for a living and would rather not do it at home as well.

The 1581 Disk Drive: A technical evaluation

M. Garamszeghy
Toronto, Ontario

. . .the 1581 supports subdirectories or, more correctly, disk partitioning, with each partition having its own BAM and directory. . .

A few days before turning the big 3–0, I received a birthday present in the form of a little plain brown box from Commodore, courtesy of Dan and Jim at Westchester. Inside, I discovered CBM's latest offering for its 8–bit line: the 1581 3½ inch disk drive. (Although the label on the front of the drive specifically says "FLOPPY DISK DRIVE", I refuse to call a 3½ inch disk "floppy", especially when my 1571 merely says "DISK DRIVE". For those not familiar with a micro disk, it is encased in a rigid plastic housing making it the least floppy of flexible disks.)

This little beast is compact, quiet, fast, and versatile. It runs on the standard or fast serial port and is primarily designed for use with the C128, but works quite nicely with the C64, Plus/4, C16 and even the VIC 20. Early prototypes were reported to have problems dealing with the slow serial bus of these other machines, but these appear to have been corrected now. The external power supply allows the drive to run much cooler than the 1541 or 1571. It still runs a bit warm, but nothing like my dual purpose toaster–cum–1541 disk drive. The physical size of the 1581 is minuscule compared to the 1541/1571 type drive.

The drive is small enough to rest nicely on the ledge behind the keyboard of my C–128. The only physical layout item which might be confusing to 1541/1571 owners is the location of the power switch. On the 5¼ inch drives, the switch is in the right rear corner. On the 1581, it is in the left rear corner. When you are used to fumbling around on one side to turn the drive on, switching sides isn't fair. On the 1581 this is readily overcome due to the small dimension. Just by putting your hand behind it, one finger is almost bound to be in contact with the switch, no matter where it is.

The 3½ inch disks are obviously also much smaller than the 5¼ floppies. Smaller yes, but they also hold more than twice as much data as a double sided 1571 disk! The built–in rigid plastic cases of 3½ inch disks make them much easier to care for, transport without the box, send in the mail, etc.

The quiet smooth operation of the 1581 is a joy (not) to listen to. In fact, with the exception of a slight click as the head steps between adjacent tracks or a low pitch squeal as it homes onto a distant track, you would hardly even know that the drive was running. Several times over the course of testing it, I thought that the 1581 had died on me only to discover a few seconds later that it was indeed alive and well and going about its normal business.

Inside, the 1581 is nicely laid out with a well shielded CHINON 80 track drive mechanism connected to the main circuit board via a plugged multi–conductor ribbon cable and a smaller dual–conductor cable. The main ribbon cable edge connector on the drive mechanism appears to be a standard type, perhaps a SASI. If this is the case, it might be possible to use the controller board with its built–in DOS to interface to other "standard" type drives such as 80 track 5¼ inch ones. Hmmm. . .

Hardware Glitch on Early 1581s

The first batch of 1581 drives have an error on the PC board that creates all sorts of problems, and can lead to the loss of data on disk. The problem is the lack of a ground connection on an IC: pin 10 of U10 (a 74LS93) should be connected to ground at pin 1 of U1 (the 6502). If you are having problems with your 1581, make this connection yourself, or bring your drive to your dealer to have the fix made under warranty. U10 is marked on the PC board, and is located near the front of the drive, on the right side of the board. Commodore is aware of the problem, and has either fixed it already, or plans to fix it on future versions of the PC board. Our thanks to Hilaire Gagne of Laurentian Business Products for bringing this problem to our attention.

Specifications

The technical parameters are summarized in Table 1. Note the difference between the "physical" and "logical" organization of

the disk. This will be explained in greater detail later. Also note the mention of directory partition. This too will be explained later. A couple of the 1581's more salient points are its capacity (800k or 3160 blocks free) and speed (average about 1.5 times as fast as a 1571 when used with a C–128 or 1.5 times as fast as a 1541 when used with a C–64, etc.). Unlike the 1571, the full capacity of the drive, along with its advanced features (except burst mode), are available to all computers capable of supporting the serial port.

Large sequential, program and user files are limited only by disk space (or available computer memory). Large relative files are fully supported up to full disk size (minus, of course, the overhead for side sectors, etc.) by the use of "super side sectors" or extended side sector blocks.

A speed test of the 1581 and comparison to other computer/drive combinations is summarized in Table 2. The tests were conducted with a C–128 running in slow (1 MHz) mode and C–64 mode using the CIA #2 TOD clock as an automatic hardware timer. The TOD (or time of day) hardware clocks are based on the system clock, and are very accurate. They are not affected by disk operations or system interrupts. Unfortunately, the TOD clocks (one for each CIA) are not used in CBM ROMware.

All functions were performed from BASIC using commands such as LOAD, SAVE, PRINT#, INPUT#, etc. The relative file test included double record positioning, although the 1581 manual assures us that this is not required for the 1581. (It is still used in the examples in the manual to "maintain compatibility with other drives".) At the start of each test, the disks contained identical sets of test files, stored in the same order. The speeds are only meant to be a relative indication of disk I/O from BASIC for a given set of conditions. Disk I/O speed depends on a number of factors such as where (i.e. proximity to directory track) and how (contiguousness) a file is stored.

As you can see, the 1581 is considerably faster in all modes of operation than either a 1541 or a 1571. The increased speed is primarily due to the track cache used by the 1581 to buffer an entire track at once in RAM for all I/O operations. After reading an entire track into RAM, any further references to that track, either reads or writes, only involve RAM–to–RAM transfer of data. The track cache is then written back to the disk if it has been changed by DOS. Incidentally, the 146 block file burst loaded in 5.3 seconds with the 1581/C–128 combination is a blistering 7000 bytes per second! It is also interesting to note that, when used with a C–64, the write speed is consistently faster than the read speed! Anyone care to hazard a guess on the reason for this?

1581 DOS

The 1581 disk operating system supports all of the standard DOS commands of other CBM drives that we have come to know and love and contains a number of refinements over these previous DOSes. Since the DOS was supposedly re– written from scratch for the new hardware, the notorious SAVE and relative file bugs are said to be finally and totally eradicated. (At least, they have not surfaced yet.) Burst mode is also supported for C–128 users. The 1581 burst command set is virtually identical to the 1571 set with a few changes to the utility commands:

"u0>b0" forces the serial bus to slow mode (default for a C–64, etc.)
"u0>b1" forces the serial bus to fast mode (default for a C–128)

These two take the place of the "u0>m0" and "u0>m1" mode commands on the 1571).

"u0>v0" turns on verify after write
"u0>v1" turns off verify after write
"u0>mr" + chr$(> memory address) + chr$(number of pages);
and
"u0>mw" + chr$(> memory address) + chr$(number of pages)

These are memory read and write commands allowing transfer of multiple blocks of data (in 256 byte pages) via burst mode. This allows direct reading and stuffing of the track cache buffer. In addition, a new switch bit has been provided for the burst read and write commands to select logical or physical track and sector numbering systems.

An extended block read and block write command set is also provided.

"b – R" (that's b – <shift> r) and
"b – W" (b – <shift> w)

These are similar to "ua:" and "ub:", except that DOS does not check to see if the track and sector numbers fall into the range that it normally expects. Presumably, this is so that you can read and write non–standard disks, maybe even weird copy protection schemes and foreign disk formats, with block read and block write.

Although it is not specifically mentioned in the User's Guide, the 1581 supports extended directory pattern matching. The pattern match character can be placed anywhere in the pattern string. For example:

"*bas" will find all files which end with "bas"
"a*b" will find all files which start with "a" and end with "b", regardless of the length of the filename.

This feature is very convenient for people who append file types, such as "bas", "ml", "text", "asm", "obj" etc. to file names. Searching for a series of assembler source files on a disk is as easy as:

directory "*.asm"

The extended pattern matching can also be used in other DOS commands such as opening or LOADing a file, scratching a series of files, etc. A nice touch, Commodore.

As previously rumoured, the 1581 supports subdirectories or, more correctly, disk partitioning, with each partition having its own BAM and directory. Each partition can thus have 296 file name entries. All this comes at a cost: each partition used as a subdirectory also requires 40 blocks of overhead for this additional BAM/directory track. In general, the partitions can be any size, from 1 logical sector up to about half of the disk capacity, and are created by a simple DOS command:

/0:{partition name}" + chr$(starting logical track#)
chr$(starting logical sector#) + chr$(< # sectors to partition)
chr$(> # sectors to partition) + ",C"

This command must be sent from the computer to the disk drive over the command channel, similar to any other BASIC 2.0 type disk operation, because neither BASIC 2.0 nor 7.0 supports a partitioning command. The partition cannot overlap the normal BAM and directory areas, hence the limitation on maximum partition size (the BAM/directory track is smack in the middle of the disk on logical track #40). The partitioning command can be used for protecting almost any size area on the disk from being overwritten by DOS. In real terms, the partition command works by assigning the disk area to a file in the root directory (you can also nest partitions but the procedure soon become very complicated) with the new file type CBM. The number of blocks indicated for the CBM file is the partitioned area size. The "file" occupies a contiguous area on the disk, and once set cannot be easily changed in size without destroying it.

To use the partition as a subdirectory area, several special steps are required. First, the starting logical sector number must be 0. Next, the partition size must be a multiple of 40 blocks (i.e. whole tracks). Third, the partition must be a minimum of 120 blocks long. After creating a partition, it can be selected by:

"/0:{partition name}"

Once selected, the partition must be formatted before first use as a subdirectory. This is done using the normal DOS format command:

"N0:disk name,ID" or
HEADER"disk name",lid

The specified disk name and ID code need not be the same as for the rest of the disk. This puts the BAM and directory information into the first track of the partition. You have to be careful with this partitioning scheme: If you format without properly selecting the desired partition, you will erase the entire disk! (It is interesting to note that the ID code is not embedded into the sector header data in the manner of 1541/1571 drives and in fact serves no real purpose on the 1581 except as a simple identifier for DOS to tell what disk it has. You can change the ID code as often as you like because it is only stored in certain areas such as the Directory header block).

Once a partition has been selected, all reads and writes to disk will be made to files in that directory only. All other files in other directories on the disk will not be found. Copies of files may reside in several partitions, even under the same names, but these are totally separate copies. The root or main directory can be selected by a:

"/0:" with no partition name specified.

A few new DOS status and error codes have been provided to indicate the selection of a partition or an illegal partition specification. If you are protecting a smaller area of the disk, say the BOOT sector, you must set up the partition BEFORE you write to that part of the disk. The partitioning process will wipe clean the area that is being protected. One thing which I do not like is that the CBM files are not "locked" entries. This means that you can wipe out an entire subdirectory with an errant SCRATCH command.

For advanced users, the internal DOS functions, such as read a sector, etc., can be accessed via a KERNAL type jump table in high ROM. All of the most important functions are also passed through individual indirect RAM vectors allowing them to be trapped and redirected. The internal DOS routines also allow you to bypass the track cache and read or write a disk sector directly via one or more of the job buffers. This handy feature frees up 5 Kbytes of drive RAM which can be used for custom drive programming. You can stuff quite a bit of ml code into 5 K!

Another feature provided is the automatic execution of an "&" type utility file on power-up or soft reset. When a reset occurs, the 1581 searches the root directory for a file named "COPYRIGHT CBM 86", then loads and executes it in drive RAM if found. This must be a USR type "&" disk utility file. It can be used to set up custom programming routines, such as changing the RAM jump vectors, automatically on drive startup or initialization.

Physical and Logical Disks

The 1581 disk is physically configured as 512 bytes per sector, 10 sectors per track, 2 sides, 80 tracks per side. It is interesting to note that the sides on the 1581 are flipped compared to the numbering system used by MS-DOS and ATARI ST disks of the same side (i.e. side 0 is side 1 and side 1 is side 0. Use the demo program at the end of this article to see this for yourself if you wish). The tracks are numbered from 0 to 79 as per the standard MFM numbering scheme, and the sectors from 1 to 10 on each side. The 512 byte physical sector size allows an extra 512 user bytes per track to be squeezed on, maximizing disk usage. The WD 1772 disk controller chip can only handle 18 x 256 byte sectors per track, equivalent to 9 x 512, at its normal recording density when you take into account the overhead bytes required for each sector. The 1581 is a radical departure from other 8-bit CBM drives in that it uses the industry standard IBM System 34 MFM recording format instead of Commodore's 4 for 5 GCR encoding scheme. This means that you can physically read and write other disk types (such as 3½ inch MS-DOS as used in many laptops and ATARI ST) in the 1581 and, perhaps more importantly, read and write 1581 disks in these other machines. Of course, you will need a suitable conversion program because the logical format of each of the above mentioned disk formats is totally different.

For some reason, probably to maintain some degree of similarity with earlier DOSes (although I don't see why because the medium is totally incompatible), the DOS uses a different logical addressing scheme. The logical scheme, which is used by all of the user DOS commands, such as block-read, etc, some of the job queue commands and some of the burst mode commands, consists of: single sided, 80 tracks (1 to 80), 256 bytes per sector, 40 sectors per track (0 to 39). In this scheme, logical sectors 0 to 19 are on side 0 and 20 to 39 are on side 1. There are two logical sectors in each physical sector. As with other versions of DOS, the first two bytes of each logical sector represent the link to the next logical sector in a given file.

The root or main directory is on logical track 40 (physical track 39). The directory header is logical sector 0, BAM for logical tracks 1 to 40 is in logical sector 2 and BAM for logical tracks 41 to 80 is in logical sector 2. The directory proper starts in logical sector 3 and uses the rest of the track. The format for partition directories is identical, except that the BAM is only local for the specific partition.

It should be noted that because the directory and BAM is in a different location on the 1581, any 1541/1571 software that accesses the directory track directly, such as unscratch, directory alphabetizing and lock-unlock programs, will not work with the 1581 without modifications.

Utilities and Documentation

The user's guide that comes with the 1581 is clearly superior to any supplied with the 1541 or 1571. It even takes the advanced programmer to heart with memory maps (albeit not very detailed) and a detailed description of how the job queue system works, complete with a listing of job codes, error returns and a BASIC example. The guide contains many more detailed examples than previous guides, but is still lacking in the area of burst mode. The burst mode descriptions are in the same weak format as the 1571 manual with no actual examples beyond a cryptic verbal description. To CBM's credit though, the demo disk that comes with the drive contains a fairly detailed example burst mode program, complete with annotated assembler source code.

The manual makes virtually no reference to the fact that the 1581 can be easily programmed to read other 3½ inch disk formats, such as IBM PC system 2 and Atari ST. It will not read Amiga or Macintosh disks directly from DOS due to a completely different physical disk structure. However, with over 5 K space available for custom programming, anything is possible.

The utility disk supplied with the 1581 comes with a number of really useful programs. The back-up routines for both the C-128 and C-64 support expansion RAM and/or multiple drives to minimize disk swapping. A fast loader, "ZAPLOAD", is provided for the C-64, which doubles the speed at which files will load. Files to load are selected from a menu. Also included are a sector editor for the C-128 (not just "display track and sector") and other utilities for creating BOOT sectors, partitions, etc. The only thing which I did not like about the sector editor was that it works on CBM DOS disks only using logical sector numbers and will not let you examine foreign disk formats very easily. The simple demo program included with this article works with any readable disk type, including MS-DOS, and ATARI ST.

A few hi-res pictures are also included in a slide show for the C-128 in 40 column mode to demonstrate the drive's speed.

Compatibility

As mentioned previously, any program that tries to access the directory track directly with block reads or writes will not find it and therefore will not work with the 1581 without modification. This includes C-64 GEOS. It will probably take some time before commercial software is supplied in the 3½ inch format and, until it is, you may have to boot up your favourite program on your 5¼ inch drive and use the 1581 for data storage only. A minor inconvenience, but generally livable. The 1571 DOS shell for the C-128 works with the 1581 in its file copying mode for transferring a range of files between drives. The various directory routines, such as unscratch and re-order, do not work for the reasons previously outlined.

Fast loaders, fast copy utilities, nibble copiers and 1541 based copy protection schemes will probably not work with the 1581 because they are too hardware and DOS specific to the older 5¼ inch drives. However, other non- copy protected software works well with the 1581. C-128 CP/M will currently not boot from the 1581, although I understand that an upgrade will be available. Once booted from a 1571 or 1541, CP/M will work, to a limited extent, with the 1581. The physical disk format is very similar to EPSON QX- 10 format. Therefore, CP/M thinks it is a QX-10 disk and treats it accordingly. The trouble with this is that you lose half of your disk capacity (QX-10 is only set up for 40 tracks). For those diddlers interested, you can use all 800 k in CP/M mode by changing a few bytes in the CPM+.SYS file disk parameter table using a debug tool such as DDT or SID. Of course, you lose compatibility with true QX-10, but who uses that format anyway?. (Complete details will be published in the next Transactor).

Demo

Listing 1 is a short demo program for the 1581. The source code for the machine language portion is contained in listing 2. The program is a simple display track and sector utility that works by direct access to the job queue and disk I/O buffers. It works with the C-128 in 80 column mode only and will read many types of 3½ inch disks. The program is very simple to use and works in terms of physical tracks and sectors rather that CBM DOS logical sectors. The operating details and command keys are contained in the REM statements and screen prompts, so I won't repeat them here.

As mentioned previously, you will find the sides flipped for non CBM DOS disks. The MS-DOS BOOT record is always on side 0, track 0, sector 1, while the 1581 finds it on side 1. Similarly, single-sided disks, such as ATARI ST, should have the data on side 0, yet the 1581 sees it on side 1. Both MS- DOS and ATARI ST use a 512 byte sector size, 9 sectors per track, numbered 1 to 9. The directory structure and disk allocation method for ATARI ST and MS- DOS are quite similar, although located in slightly different areas of the disk.

Since the demo program uses physical track and sector numbers, the CBM DOS directory will be on side 0, track 39. The directory proper starts at physical sector 2½ (i.e midway through sector 2) and continues to the end of side 1, track 39, sector 10. The C-128 BOOT sector is located in the first half of side 0, track 0, sector 1.

Final Word

The 1581 is well worth its long delay in getting to market but, like some other products, its usefulness is limited by the lack of software available for it. In all fairness, I must give the guys at CBM credit for doing an excellent job on this one. Its large disk capacity is a breath of fresh air, especially for C-64 users who have frequently complained about the 1541's puny 170 K capacity.

Table 1: 1581 Technical Specifications

Total formatted capacity	808,960 bytes
Number of directory entries	296 (each directory partition)
Maximum SEQ file size	802,640 bytes
Maximum REL file size	800,000 bytes (approx)
Max. records per REL file	65,535
Number of DOS RAM buffers	9 (7 I/O + 2 reserved for BAM)
Track cache buffer	5 K bytes
Recording format	MFM (IBM system 34)

Physical Disk Organization: (as seen on the disk)

Number of sides	2 (numbered 0 and 1)
Number of tracks per side	80 (numbered 0 to 79)
Number of sectors per track	10 (numbered 1 to 10)
Number of bytes per sector	512

Logical Disk Organization: (as seen by DOS)

Number of sides	1
Number of tracks	80 (numbered 1 to 80)
Number of sectors per track	40 (numbered 0 to 39)
Number of bytes per sector	256
Number of blocks free	3160

Notes:
All logical to physical conversions are done automatically by DOS. Each physical sector is subdivided into 2 logical sectors. Each logical sector begins with the track and sector pointer to the next logical sector, as per normal CBM DOS.)

Chips:

Microprocessor	6502 A
I/O Interface	8520 A
32 K bytes ROM	23256
8 K bytes RAM	4364
Disk controller	WD 1772

Physical Dimensions:

Height	63 mm
Width	140 mm
Depth	230 mm
Weight	1.4 kg

External Power Supply:

North America	100–120 VAC, 60 Hz, 10 W
Europe	220–240 VAC, 50 Hz, 10 W

Table 2: Summary of Disk Drive Test Speeds

Operation	Disk Drive / Computer			
	1541/64	1571/128	1581/128	1581/64
FORMAT disk	80	40	100	100
LOAD short	4.5	1.5	0.4	3.5
LOAD long	93	11.5	6.4	75
ZAP LOAD long	30			
Burst LOAD short		0.4	0.4	
Burst LOAD long		10.4	5.3	
SAVE short	6.5	5.7	2.1	4.5
SAVE long	100	71	31.5	52
WRITE SEQ file	9.3	7.8	5.7	6.8
READ SEQ file	8.1	4.6	3.8	7.3
WRITE REL file	155	109	57.3	89.5
READ REL file	80	77	32.3	37.5

Notes:

1) All times are in seconds and were obtained using the CIA #2 TOD clock as a timer. The times you get may vary depending on disk usage (i.e if the files are stored contiguously or not, and how far the head has to move to access the files).

2) The short program is 5 blocks in size and the long is 146 blocks.

3) ZAPLOAD 64 is a fast loader for the C–64 and 1581 which is supplied on the 1581 demo disk. It takes 6 seconds to load.

4) The SEQ file consists of 100 strings of 24 characters each. File reads and writes are via BASIC's INPUT# and PRINT# statements in a FOR–NEXT loop.

5) The REL file consists of 100 records of 64 bytes long each. Records are read and written in BASIC in pseudo–random order (100, 1, 98, 3,. . . , 99, 2) designed to maximize record searching. Record positioning commands given twice for each record. The write speeds include initial creation of a 64 byte x 100 record file.

Listing 1: 1581 drive demo program for the C128.

Note: With the exception of "[N spcs]", literals such as "<left>" and "<up>" that appear in this program should be entered AS SHOWN – do NOT replace these with cursor control characters.

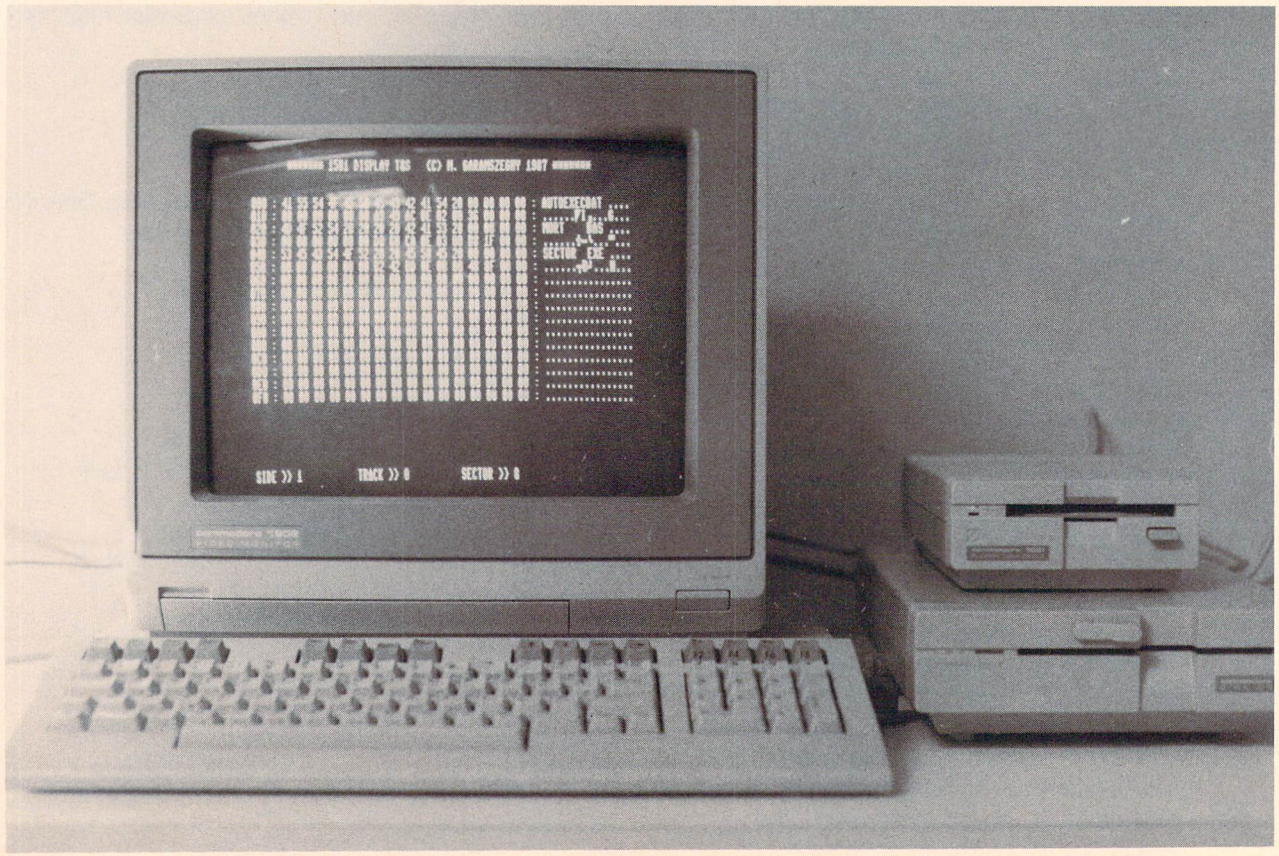
```
GH 1000 rem ********************************
GD 1010 rem *                              *
GB 1020 rem *          1581 demo program   *
HF 1030 rem *                 by           *
MB 1040 rem *            m. garamszeghy    *
OF 1050 rem *                              *
AO 1060 rem *       for c-128 and 1581 drive *
CH 1070 rem *                              *
GM 1080 rem ********************************
GL 1090 :
JM 1100 dv=9          : rem disk drive device number
KM 1110 :
FH 1120 fast:print chr$(147);:b1=3328:b2=3338
FP 1130 for i=2816 to 2935:read x:poke i,x:next
        : rem poke machine code
IO 1140 :
MH 1150 print"[7 spcs]******* 1581 display t&s  <c> m.
        garamszeghy 1987 *******"
MP 1160 :
KP 1170 sp$="s":print:print:input"[s]creen or [p]rinter";sp$
HK 1180 pd=3:if sp$="p" then pd=4
NN 1190 open 1,pd: rem open output file
EC 1200 :
PG 1210 print:print"insert disk then press a key . . ."
        :getkey i$
DJ 1220 open 15,dv,15,"u0"+chr$(10): sys 2816,0
        : rem analyse side 0
PE 1230 print#15,"u0"+chr$(26): sys 2816,10
        : rem analyse side 1
ME 1240 :
LF 1250 t1=peek(b1) and 14 :t2=peek(b2) and 14
        : rem burst status byte side 0 and 1
KE 1260 if t1 and t2 then print "*** disk error ***"
        :getkey a$:goto 1720
DK 1270 sz=peek(b1) and 48: if t1 then sz=peek(b2)
        and 48: rem check sector size
```

```
FM 1280 sz=sz/16:if sz=3 then sz=4
HF 1290 s1=peek(b1+4):m1=peek(b1+5)
NG 1300 s2=peek(b2+4):m2=peek(b2+5)
JC 1310 window 0,2,79,24,1:if pd=4 then cmd 1
       :gosub 1940:print#1:close1:open1,pd
NE 1320 gosub 1940
NP 1330 si=-1:input"side,track,sector";si,t,s:if si=-1
       then 1720
EI 1340 print#15,"m-w"chr$(206)chr$(1)chr$(1)chr$(si)
       : rem set side
JG 1350 print#15,"m-w"chr$(11)chr$(0)chr$(2)chr$(t)chr$(s)
       : rem set t & s
EM 1360 :
CK 1370 rem then stuff job code for read physical sector
        into job buffer
LN 1380 print#15,"m-w"chr$(2)chr$(0)chr$(1)chr$(164)
HO 1390 gosub 1410: goto 1440: rem wait till job done
MO 1400 :
HJ 1410 print#15,"m-r"chr$(2)chr$(0)chr$(1) : rem check
       job status
FD 1420 get#15,a$:if asc(a$)>127 then 1410 : else return
KA 1430 :
NI 1440 window 0,24,79,24,1:print#1,"side >>"si,
       "track >>"t,"sector >>"s;
MK 1450 window 0,20,79,23
MO 1460 print"<left> - decrease sector#[11 spcs]
       <right> - increase sector#"
LA 1470 print"<down> - decrease track#[15 spcs]
       <up> - increase track#"
GE 1480 print" <esc> - select new track, sector[3 spcs]
       <space> - next 256 bytes ";
CE 1490 if asc(a$)>1 then print#1,"*** error #"asc(a$)
       "***":goto 1320
AF 1500 :
OP 1510 print#15,"u0>mr"chr$(3)chr$(sz)
       : rem burst mode memory read
CD 1520 sys 2819,sz :rem read data via burst mode
OG 1530 :
AA 1540 if pd=4 then print#1," "
DA 1550 for i=4864 to 4864+sz*256-1 step 256
       : rem display data in hex form
IK 1560 window 0,2,79,19,1
GJ 1570 for j=0 to 254 step 16 : rem 16 bytes per line,
       256 bytes per page group
AK 1580 :
JL 1590 nu$=right$(hex$(i+j-4864),3)+" : ":n2$=": "
LA 1600 for k=0 to 15:n2=peek(i+j+k)
       :nu$=nu$+right$(hex$(n2),2)+" "
CG 1610 n$=chr$(n2):if n2<32 or (n2>127 and n2<160)
       then n$="."
ID 1620 n2$=n2$+n$:next:print#1,nu$;n2$:next
CN 1630 :
GI 1640 if pd=3 then getkey a$ : rem get a key press
CA 1650 if a$=chr$(145) then t=t+1: goto 1340
       : rem cursor up = increase track #
CC 1660 if a$=chr$(17) then t=t-1: goto 1340
       : rem cursor down = decrease track #
BN 1670 if a$=chr$(157) then s=s-1: goto 1340
       : rem cursor left = decrease sector #
HK 1680 if a$=chr$(29) then s=s+1: goto 1340
       : rem cursor right = increase sector #
AC 1690 if a$=chr$(27) then 1310
       : rem escape = select new values
BL 1700 next:goto 1310
       : rem other key = display next group
CC 1710 :
EA 1720 dclose u(dv):close 1:window 0,0,79,24,1:end
GD 1730 :
EK 1740 rem machine code data
KE 1750 :
FD 1760 data 76, 44, 11, 76, 84, 11, 169, 0
HH 1770 data 141, 0, 255, 96, 169, 8, 44, 13
PO 1780 data 220, 240, 251, 173, 0, 221, 73, 16
FO 1790 data 141, 0, 221, 173, 12, 220, 96, 120
PB 1800 data 44, 13, 220, 173, 0, 221, 73, 16
AA 1810 data 141, 0, 221, 96, 133, 250, 32, 6
KO 1820 data 11, 169, 13, 133, 251, 32, 31, 11
DD 1830 data 160, 0, 32, 12, 11, 145, 250, 41
JD 1840 data 14, 208, 15, 200, 192, 2, 208, 242
AP 1850 data 32, 12, 11, 145, 250, 200, 192, 7
GI 1860 data 208, 246, 88, 96, 133, 252, 169, 0
HJ 1870 data 133, 250, 169, 19, 133, 251, 32, 6
HD 1880 data 11, 32, 31, 11, 160, 0, 32, 12
IK 1890 data 11, 145, 250, 200, 208, 248, 198, 252
GJ 1900 data 240, 224, 230, 251, 76, 102, 11, 255
KO 1910 :
NG 1920 rem print disk details
OP 1930 :
FI 1940 print "sector size ="sz*256:print:if t1 or t2 then
       print "single sided!!"
ED 1950 if t1 then 1970
BL 1960 print "side 0:":print "min sector #";s1,"max
       sector #"m1
EF 1970 if t2 then 1990
EG 1980 print:print "side 1:":print "min sector #";s2,"max
       sector #"m2
ON 1990 print: return
```

Listing 2: Source code (in "Buddy" or "PAL" assembler format) for the machine language code used in the demo program.

```
100 ; 1581 display t&s burst mode source code
110 ;
120 ; <c> m. garamszeghy 1987
130 ;
140 ;
150 pntr    =   $fa        ;zero page pointer
160 size    =   $fc        ;number of blocks to read
170 mmu     =   $ff00      ;mmu config reg
180 data    =   $1300      ;start of data buffer
190 conf    =   $0d00      ;start of disk type buffer
200 clock   =   $dd00      ;burst clock line
210 c1dr    =   $dc0c      ;burst data register
220 c1icr   =   $dc0d      ;burst interrupt register
230 ;
240 ;
250 *=$0b00                ;decimal 2816
260 .opt oo                ;assemble to memory
270 ;
280 jmptable =  *
290         jmp  testdsk   ;analyze disk
```

```
300             jmp  readdata   ;burst memory read
310 ;
320 setbnk      =    *          ;set to bank 15
330             lda  #0
340             sta  mmu
350             rts
360 ;
370 toggle      =    *          ;toggle data line
380             lda  #8
390 tog1        bit  c1icr
400             beq  tog1       ;wait for icr
410             lda  clock
420             eor  #$10       ;toggle clock
430             sta  clock
440             lda  c1dr       ;get a data byte
450             rts
460 ;
470 reset       =    *          ;init burst mode
480             sei
490             bit  c1icr
500             lda  clock
510             eor  #$10       ;toggle clock
520             sta  clock
530             rts
540 ;
550 testdsk     =    *          ;burst mode query disk format
560             sta  pntr       ;save offset into format buffer
570             jsr  setbnk     ;go to bank 15
580             lda  #>conf     ;high byte of format buffer
590             sta  pntr+1
600             jsr  reset      ;start burst mode
610             ldy  #0
620 tes1        jsr  toggle     ;get first status byte
630             sta  (pntr),y   ;and stash it
640             and  #$0e       ;check for errors
650             bne  exittest
660             iny
670             cpy  #2
680             bne  tes1       ;get next status byte
690 tes2        jsr  toggle     ;get data byte
700             sta  (pntr),y   ;and save it
710             iny
720             cpy  #7         ;check for all done
730             bne  tes2
740 exittest    =    *          ;exit disk test routine
750             cli             ;restore interrupts
760             rts
770 ;
780 readdata    =    *          ;burst mode memory read
790             sta  size       ;number of pages to read
800             lda  #<data
810             sta  pntr       ;set pointer to data buffer
820             lda  #>data
830             sta  pntr
840             jsr  setbnk     ;set bank to 15
850             jsr  reset      ;start burst mode
860             ldy  #0
870 rea1        jsr  toggle     ;get data byte
880             sta  (pntr),y   ;and save it
890             iny
900             bne  rea1       ;end of page?
910             dec  size
920             beq  exittest   ;last page?
930             inc  pntr+1
940             jmp  rea1       ;go get next page
950 ;
```



CP/M and the 1581 Disk Drive

M. Garamszeghy
Toronto, Ontario

Upgrade your C128 CP/M for the new CBM drives

Older versions of C–128 CP/M will not fully support the new 1581 disk drive. This is a pity because the large capacity of the 1581 combined with its high speed make it an ideal CP/M drive. Of course, you can mail in the coupon which comes with the 1581 along with some of your hard earned cash and get yet another "upgrade" version of CP/M which does support the 1581. I already have three different versions of CP/M for my C–128 (the original plus two "upgrades") so why do I need another one just to use the 1581? The short answer is that I do not.

The first problem involves creating a 1581 CP/M format disk. This is easiest to do with the C–128 in native mode using the burst mode format command since older versions of CP/M's FORMAT.COM utility do not work properly with the 1581. The following command string can be used, assuming your 1581 is device 9:

```
open15,9,15
print#15,"u0"+chr$(134)+chr$(2)+chr$(79)+chr$(10)
  +chr$(0)+chr$(229)+chr$(1)
close 15
```

This will format the disk identical to a 1581 DOS disk physical structure (i.e. 512 byte/sector, 10 sectors track) but without the directory, BAM, etc. and fill it with CP/M blank disk bytes ($e5 or dec 229).

Next, with a few simple modifications to the CPM+.SYS file, you can take full advantage of the capacity of the 1581 without forking out for the latest upgrade disk. You can still use the disk formatted by the method above with CP/M if you do not make the following mods, but you will only be able to fill it half full. This mod will not allow you to boot CP/M from the 1581, but it will allow you use the 1581 once CP/M has been booted from a 1571 or 1541. (In my present set up, the 1571 is device 8 and the 1581 is device 9. CP/M can only be booted from device 8 anyway). A note in Commodore's favor is that the CP/M upgrade will allow you to boot from the 1581 if it is connected as device 8.

The procedure involves changing a few bytes in part of the CPM+.SYS file know as the "disk parameter block table". This table, which is described in detail in the "CP/M 3 System Guide" under section 3.3 BIOS Data Structures on pages 40 to 44, contains the data for the physical characteristics of the MFM disk formats supported by C–128 CP/M. The location of the table in the CPM+.SYS file depends on the version of CP/M that you are using. The instructions for all three current versions are outlined below. The locations are summarized in Table 1 and referenced in the text.

Before continuing, you will need a formatted CP/M disk (C–128 1541 single or 1571 double sided format) containing the CP/M system files (CPM+.SYS and CCP.COM) and the utility SID.COM (or the older DDT.COM or equivalent debugger utility). You can also include a copy of SHOW.COM to check the results afterwards. The SID.COM program resides on the CP/M additional utilities disk that comes with the Digital Research CP/M plus documentation. Several other public domain debuggers are also available if you do not have SID.

**NOTE: USE A BACK UP WORK DISK.
DO NOT DO THIS WITH YOUR ORIGINAL SYSTEM DISK BECAUSE IT WILL MAKE PERMANENT CHANGES TO THE OPERATING SYSTEM.**

After booting up CP/M, note the version date printed on the screen. This will be used as a reference for the addresses in Table 1. Insert your work disk and type in:

```
sid cpm+.sys   <return>
```

Note that for clarity, I will use lowercase letters to indicate items that you type in and UPPERCASE for prompts made by the computer.

After a few moments, the following status message will be displayed:

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

where zzzz is a hexadecimal number based on the version date, as listed in Table 1. Jot this down for future reference as it will be needed when saving the changes. The "#" symbol is SID's command prompt.

Now we are ready to make the changes. The physical format of a 1581 disk is identical to that of the EPSON QX–10 (10 sectors

per track, 512 bytes per sector, 2 sides). The only difference is in the number of tracks per side (80 for the 1581 compared to 40 for the EPSON). It makes things easier to start with the EPSON parameters and change them a bit rather than create a whole new entry. (The modifications will still permit full read compatibility with the EPSON disks and nearly full write compatibility on the 1571. The incompatibility in writing is only that the operating system may attempt to write more to the EPSON disk than it can hold, thus causing a disk error if you try to write to an almost full disk).

Next type in:

```
dyyyy  <return>
```

where yyyy is taken from Table 1. This is SID's d or display memory command. Note that in this, and all other SID commands, there is no space between the command letter and the command parameters. For example, d yyyy would produce an error prompt.

SID will respond with a display similar to:

```
y y y y: 50 00 04 0F 01 BD 00 7F 00 C0 00 20 00 02 00 02 P. . . . . .  . . . .
y y mm: 03 0A 45 70 73 6F 6E 20 51 58 31 30 49 A5 00 81  . . Epson QX10 I . . .
```

plus some more rows of similar hexadecimal numbers followed by the "#" prompt. The next step is to change some of the bytes. The ones that are changed are referred to in Digital Research's documentation as EXM (extent mask – 1 byte), and DSM (total drive storage – 2 byte word). These are the two parameters which tell the system how much data it can store on a disk. The change is done with SID's s or set memory command. Type in:

```
sxxxx  <return>
```

where xxxx is taken from Table 1. SID will respond with:

```
xxxx  01
```

The 01 is the current value of the byte at this location. Change it to the desired value by typing in 00 followed by return. SID will then prompt for the next byte:

```
xxx1  BD
```

Type in 86 followed by return. SID will then prompt for the next byte:

```
xxx2  00
```

Change this by typing in a 01 followed by return. That completes the major changes. At the next prompt

```
xxx3  7F
```

Enter a period "." followed by return. This should bring back the main SID prompt "#". If you want to change the disk type name from Epson QX10 to say 1581, use the s command again:

```
snnnn
```

When SID prompts with:

```
nnnn  45
```

Type in a quote (") followed by 3 spaces then 1581 and 3 more spaces and a return. At the next prompt, type in a period to return to the main SID prompt. Type in: dyyyy <return> again to check your changes and the following should be displayed:

```
y y y y: 50 00 04 0F 00 86 01 7F 00 C0 00 20 00 02 00 02 P. . . . . .  . . . .
y y mm: 03 0A 20 20 20 31 35 38 31 20 20 20 49 A5 00 81  . .     1581     I . . .
```

The final step is to save the changes. This is done with SID's w or write command:

```
wcpm+.sys  0100  zzzz  <return>
```

Once this has been done, your new CP/M system is ready for action. Re–boot the computer (you need to boot the modified CP/M system) and turn on your 1581 as drive b: (device 9). You can now PIP some files to a formatted 1581 disk (as outlined above) and use it at will for all file storage. You can check the capacity of the drive by using the SHOW.COM utility:

```
show b:
```

will display the amount of space currently available on the 1581. If it does not give something like 800k read write space free with a formatted blank disk, then your changes may have gone awry. Double check the changes outlined above and try again.

The procedure outlined above can also be done using the RAM disk (drive m:) as the working drive if you have a 1750 RAM expander. In this case, you would PIP the required files to the RAM disk, do the modifications and PIP them back again to your work disk. The same method can be used to alter the disk parameter table to support other CP/M disk formats on the 1571, such as Televideo, Xerox, DEC, etc which are not normally supported by C–128 CP/M. In this case, I would suggest that you consult the explanation of the table parameters in the CP/M System Guide. You also need to have a thorough understanding of the disk format that you wish to implement.

Table 1: Summary of CPM+.SYS file addresses

	Values by CP/M Version Date		
Parameter*	1 Aug 85	6 Dec 85	8 Dec 85
zzzz	5d00	6400	6400
yyyy	1400	2161	2161
xxxx	1404	2165	2165
nnnn	1412	2173	2173

Note: Refer to text for explanation of parameters

Programming The 1541

Frank DiGioia
Stone Mountain, GA.

. . .Now, even the weekend hacker can enjoy the thrills and chills of being in complete control of the disk drive. . .

There was a time when the idea of programming the Commodore 1541 disk drive was little more than a whimsical desire of late night, die-hard hackers. But now, programs ranging from word processors to new operating systems are utilizing the free RAM in the 1541 to reprogram the drive, allowing it to perform tricks no one previously would have dreamed possible. A wealth of information concerning the inner workings of the 1541 has become available to the public. Now, even the weekend hacker can enjoy the thrills and chills of being in complete control of the disk drive. This article is intended primarily for those who just want the chance to try some easy-to-type-in experiments with their 1541 disk drives. I am, however, including some detailed technical information so that those readers who desire to experiment beyond what is presented in this article may be better equipped to do so. In the following paragraphs I will provide brief coverage of the Job Queue, the M-R command, the M-W command and the M-E command. We will wrap things up with an example machine language program you can execute in your 1541 (if you dare!).

The Memory Commands

We will start our adventure with the memory commands. Each of the memory commands (in fact, ALL of our communications with the drive) will be sent over the error channel #15 (I'll always use file number 15 in my examples). These commands are all covered in your 1541 manual, but we will focus here on their use in programming the disk drive. Let us begin with the Memory-Read (M-R) command. The M-R command allows you to peek at the contents of the disk drive's internal memory - both RAM and ROM. To use this command you simply give the address (in lo/hi format) of the first byte you wish to read. If you want to read more than one byte, the address must be followed by the number of consecutive bytes (up to 255) you wish to receive. If you are working from BASIC these parameters must be sent as character codes. The general format of the command is

```
PRINT#15,"M-R"CHR$(lo)CHR$(hi)CHR$(n)
```

Once you issue the command to the drive, you can read the result over the error channel, via GET#, as character data. Here is a useful subroutine which utilizes the M-R command to tell weather or not a disk is write protected. You might use this routine in your own programs in order to warn the user to write-protect the program disk or to unwrite-protect the data disk. The routine works by examining bit 4 of location $1C00 (7168) in your drive's memory. This location is the main control register for the 1541.

```
400 OPEN15,8,15,"M-R"+CHR$(0)+CHR$(28)
410 GET#15,A$:A=ASC(A$+CHR$(0)):CLOSE15
420 WP=A AND 16:RETURN
```

If WP = 0 then the disk is write protected.

The Memory-Write (M-W) command is used to poke data into the 1541's RAM. This command is used to set values in various memory locations and to transfer programs from the C64 to the 1541. The general format of the command is

```
PRINT#15,"M-W"CHR$(lo)CHR$(hi)CHR$(n)<data>.
```

As you can see, this command uses the same parameters as the M-R command (you may NOT omit the length parameter) except you additionally place the data to be sent on the command line. The data may be a single character code or a string of up to 34 characters. The following routine will turn the 1541's motor on by READING the contents of the main control register, SETTING BIT 2 and WRITING the result back to the control register. The motor is turned off again by clearing bit 2.

Turn motor on:

```
10 OPEN15,8,15,"M-R"+CHR$(0)+CHR$(28)
20 GET#15,A$:A=ASC(A$+CHR$(0))
30 X=A OR 4
40 PRINT#15,"M-W"CHR$(0)CHR$(28)CHR$(1)CHR$(X)
50 CLOSE15
```

To turn the motor off, replace line 30 with:

```
30 X=A AND 251
```

You may have noticed that both of the routines above work by reading or changing a bit in the control register located at $1C00 in the 1541. Since this is a fairly important register I will interrupt myself for a moment to list what each bit in the register does.

REG at $1C00 BITS: 76543210

BIT 0 – Bits 0 and 1 are used to move the drive head
BIT 1 – See example program at end of article.
BIT 2 – Turn motor on/off
BIT 3 – Turn red light on/off
BIT 4 – Write-protect indicator
BIT 5 – Recording density bit 1
BIT 6 – Recording density bit 2
BIT 7 – Sync byte indicator

Okay. . . back to the memory commands:

The MEMORY EXECUTE (M-E) command is like a SYS command for any disk drive routine in ROM or RAM. Simply give the address of the routine in LO/HI format. The following example will execute the ROM routine which initializes the diskette in the drive. It has the same effect as typing OPEN15,8,15,"I".

```
10 OPEN15,8,15:PRINT#15,"M-E"CHR$(5)CHR$(208)
   :CLOSE15
```

Digging Deeper

The memory commands are nice for little tricks such as the ones in the example programs above. When it comes to running "real" jobs in your drive, however, it is often convenient to use the job queue. Some people tend to be scared off by the mention of the job queue, but let me try to explain how jobs are executed in the 1541 and you will then see that using the job queue is really just a shortcut to programming the disk drive.

Allow me to present three short paragraphs of background material before we jump into using the job queue.

The 6502 in your disk drive has a split personality. Part of the time it is acting as the interface processor (IP) and part of the time it is acting as the floppy disk controller (FDC). For the sake of brevity, I will just refer to the IP and FDC as though they were two completely different processors running at the same time, since this is the illusion that the designers of the 1541 intended to create. (*In fact, in earlier Commodore dual drives like the 4040 and 8050, there* were *two separate processors – the 1541's DOS is closely based on the DOS from the early drives. –Ed.*)

The interface processor is primarily responsible for those tasks having to do with communication between the computer and the disk drive, and between the disk drive and the user. In other words, it is responsible for parsing disk commands, transferring data over the serial bus and operating the red light on the front of your drive. In addition to its communication jobs, the IP is also responsible for the "soft" side of file management (ie. allocating buffers, opening files, keeping track of the BAM, etc).

The floppy disk controller, on the other hand, concerns itself primarily with those tasks that are directly related to reading or writing to the disk, or which directly control the hardware of the drive. These tasks include moving the drive head, coding/decoding GCR data, reading and writing GCR data from/to the disk surface, formatting a diskette and various other hardware related tasks. (Technical note: GCR stands for Group Coded Recording. All data is converted to GCR format before being written to the disk surface. The GCR coding scheme insures that no data will ever produce bit patterns capable of confusing the disk hardware).

The Job Queue

Does it sound as though the FDC routines might be complex or difficult to use? Well, for the most part, they are. Fortunately for us, there is an easy way to get at the FDC routines and that is through the JOB QUEUE.

About 100 times per second, the FDC scans the job queue to see if there is any work for it to do. When you give the drive a command, the IP parses it and breaks it up into a series of JOBS. It drops the jobs into the job queue and the FDC executes them. Bossing the FDC around is as easy as simply dropping a number into the job queue. Here's what you need to know:

The JOB QUEUE is simply locations $00 to $05 in your disk drive's zero page memory. We will only concern ourselves with the first four positions ($00 to $03). Associated with each position in the job queue are two bytes that tell the FDC which track and sector the job should act on. Also associated with each of the first four positions in the job queue is a data buffer that is to be used by the job. I think now would be a good time for a table:

Job Queue Position	Location of TR/SE Info	Location of Data Buffer
$0000	$0006/$0007	$0300–$03FF
$0001	$0008/$0009	$0400–$04FF
$0002	$000A/$000B	$0500–$05FF
$0003	$000C/$000D	$0600–$06FF

There are only seven job codes to choose from when using the job queue.

HEX Code	DEC Code	Job Code Description
$80	128	Read a sector
$90	144	Write a sector
$A0	160	Verify a sector
$B0	178	Seek a track
$C0	192	Slam head against end stop
$D0	208	JMP to start of buffer
$E0	224	Execute program in buffer

To use the job queue you just have to make three decisions: (1) Which Job should I run? (2) Which buffer should it use? (3) Which track and sector should it act upon?

It is important that you realize that the FDC will begin executing your job the moment you place the job code into the queue. It is therefore important that you place the track and sector information (when required) into the proper position of the TR/SE table *before* dropping a code into the job queue.

When the FDC finishes the job you asked it to perform, it writes a RETURN CODE in the same job queue position where you left the job code. Job codes are all greater than or equal to $80 (128) and return codes are always less than $80. This distinction is how the FDC tells whether a number in the queue is a job request or a return code. You can use the return code for two purposes. First of all, it tells you the job has completed (when a number less than 128 is in the queue) and secondly, it tells whether or not the job completed successfully. If the return code is 1, the job was successfully completed – any other return code indicates that the job was not successful.

The main advantage (and danger) in using the job queue is the fact that no error checking is done on the track and sector you supply in the TR/SE table. This means that you can read and write beyond the normal 35 tracks of a diskette. While I have not been too successful in my attempts to write data beyond track 35 (probably because I did a sloppy job of formatting these tracks) I have had success in READING data beyond track 35. The BASIC program called READ ANY SECTOR found at the end of this article will allow you to read data from any sector of a disk from track 1 to track 40. I have used this program to read "secret" data on some protected diskettes. The program is capable of reading "under" errors and will allow you to view data from any sector on your disk whether the sector contains ASCII data, screen codes or machine language. The program will 'hang', however, if you try to read a track that has not been formatted at all.

Executing Programs In The Drive

The last topic I would like to cover in this article is not for the faint hearted. It concerns actually programming the drive beyond simply using the job queue. It should be mentioned here that any mistakes made at this level of programming could easily destroy data on any disk that happens to be in the drive and could, in fact, cause harm to your drive itself.

The main power of the job queue, as mentioned earlier, is that you can do normal tasks (reading, writing, etc) without being limited by error checking routines. You are still limited, however, to the seven jobs listed above. If you want to do something a bit more radical, such as writing a new formatting routine, creating exotic protection techniques and so on, you will find that the job queue does not afford you the power that you need. You will have to go on to the less–travelled path of programming the disk drive. Before you can even begin thinking about programming the disk drive, however, you need a complete zero–page memory map of the drive and a listing of the ROMs. (The Transactor plans to publish Jim Butterfield's commented Disassembly of the 1541 and 1571 ROMs in book form at some time in the near future. If you are impatient, there are several other good books currently available as I'm sure every reader is aware.)

As an example of programming the disk drive, I wrote a program called DISK DESTROYER. I chose this program because it is simple, short and can hardly fail to work. As the name implies, this program will destroy whatever disk happens to be in the drive when the program is run. The program will not physically harm the diskette but will scramble the contents so badly that the drive light will not even come on when you try to load a directory from it. This program is useful mainly for data security. It will completely purge a disk in about 10 seconds (compared to a minute and a half to format a disk) and will not leave a trace of the original data. This capability can be useful for destroying old data disks or for creating a program which can only be run once. As an example of the latter, suppose you wish to have a friend try out a program you are writing. If you do not want your friend to be able keep a copy of the program, you can hide disk destroyer in an unused sector on the diskette (see the program DEATH SECTOR at the end of this article). At the end of your boot program you can execute disk destroyer with a BLOCK–EXECUTE command and disk destroyer will wipe out the disk in a matter of seconds.

Disk Destroyer

Disk Destroyer works by stepping the drive head back to track one and then calling a ROM routine which effectively destroys the track. Disk Destroyer then moves the head to the next track and repeats itself until it reaches the ending track. You should note that the current track number is stored in location $22 and that the drive head is moved by rotating bits 1 and 2 of the main control register ($1C00).

At the end of this article you will find a BASIC listing called DEATH SECTOR and the commented assembly language source code for Disk Destroyer. DEATH SECTOR will write Disk Destroyer onto any sector of your diskette in a form which can be executed with the Block–Execute (B–E) command. After the sector is written, DEATH SECTOR will print a command line on your screen which can be used to destroy the disk. If you wish to destroy the disk immediately, move your cursor to the line which DEATH SECTOR just printed and hit return. If you don't wish to destroy the disk right now, write down the line that DEATH SECTOR printed. This command line can be used at any time to destroy your disk in a matter of seconds. The format of the BLOCK–EXECUTE command to destroy a disk which contains a death sector is as follows:

```
OPEN15,8,15:OPEN2,8,2,"#0":PRINT#15,
"B-E"2;0;T;S:CLOSE2:CLOSE15
```

T and S are the track and sector where DISK DESTROYER is hidden on the disk. Note: It is very important that you open buffer "#0" rather than the usual OPEN2,8,2,"#".

Warnings & Hints

Although I don't wish to scare off any potential 1541 programmers, you need to understand, before attempting to program your drive, that it is possible to seriously harm your drive if you aren't careful (and perhaps even if you are) with your work. Although the programs presented here have been tested on my equipment, you should be careful the first time you run them on yours in case you made an error in typing or in case an error was made in typesetting this article. In any case, neither I nor The

Transactor can accept any liability for damages of any type resulting from the use or misuse of these routines. Therefore you are well advised to be certain that you understand what is going on before using any programs presented with this article.

Here are some guidelines which may save you a hefty repair bill in case things DO go wrong when you are programming your disk drive. First of all, keep your hand on the disk drive's power switch when testing a program. TURNING OFF THE COMPUTER WILL NOT STOP THE PROGRAM EXECUTING INSIDE THE DRIVE. You must turn the drive itself off to stop it. Secondly, if something DOES go wrong and the drive doesn't seem to work properly try typing

OPEN15,8,15,"I":CLOSE15

while the drive's door is open and no disk in the drive. Then put a disk that you know to be good (but make sure you have a backup of the disk) into the drive and see if the drive can read it. It is quite likely that this will fix any problems you may encounter provided you have a quick hand on the drive switch.

In writing and testing Disk Destroyer, I made two mistakes – one that sent the drive head off searching for track 100 and another which caused it to try to step backwards 255 tracks. Neither of these errors caused any damage to my drive although the repeated knocking caused by the drive head trying to move outward 255 tracks could have sent me to the repair shop if I had not quickly turned off the drive when I heard the buzzing sound of the drive head hammering against the end stop. The attempt to find track 100 simply required me to use the line of code given above to initialize the drive – pulling the read/write head back into familiar territory.

One last word of wisdom is to turn off your stereo, drive fan or whatever noisy appliance happens to be running while testing your drive routines since sound is your key indicator of something going wrong. Each time the head moves by one track, you should hear a slight clicking sound. If you hear it move more than 40 tracks you know something must be wrong. Likewise, if you hear the head slamming against the end stop you know to switch off the drive FAST.

While you should be aware of the potential to damage your equipment, it is my experience that a quick hand on the drive's power switch should make programming your disk drive almost risk free.

Listing 1: This program will create a "death sector" on the track and sector of your choice on the disk currently in the drive, and show you the command required to activate it. When the command is given at any time when that disk is in the drive, the contents of the disk will be completely destroyed within ten seconds. This provides an easy way to destroy sensitive data in a hurry. (We call it "The Oliver North Special".)

The actual program that the drive executes to destroy the disk can be seen in assembler form in Listing 3.

```
CK 100 rem*********************************
OM 110 rem              death sector
FP 120 rem            frank e. digioia
AM 130 rem*********************************
CK 140 rem      the transactor vol 8 issue 3
EN 150 rem*********************************
JA 160 data"warning!!! this program will "
PG 170 data"create a sector on your disk "
IH 180 data"which will cause the disk to "
LC 190 data"self-destruct if b-e command"
CN 200 data"is issued on that sector. "
AB 210 rem*********************************
PH 220 read a$,b$,c$,d$,e$:rem read warning
MM 230 r$=chr$(13):print a$r$b$r$c$r$d$r$e$
FD 240 gosub570:rem verify disk name
NG 250 open2,8,2,"#0":rem open buffer 0
KM 260 print#15,"b-p";2;0:rem start of buf
NF 270 for i=1 to 146:read ml:ck=ck+ml
DK 280 print#2,chr$(ml);:next:rem fill buf
DE 290 data 234, 234, 234, 169,   1, 133,   6, 169
OC 300 data   0, 133,   7, 173, 142,   3, 174, 143
GC 310 data   3, 172, 144,   3, 141,   0,   3, 142
IE 320 data   1,   3, 140,   2,   3, 169, 224, 133
OO 330 data   0, 165,   0,  48, 252,  96, 165,  34
KI 340 data 240,   4, 201,  36, 144,   4, 169,  46
KM 350 data 133,  34,  32, 102,   3,  32, 102,   3
EC 360 data 198,  34, 208, 246, 230,  34,  32, 163
HF 370 data 253,  32,   0, 254, 165,  34, 201,  36
GJ 380 data 176,  11, 230,  34,  32, 109,   3,  32
CI 390 data 109,   3,  76,  62,   3, 173, 141,   3
GA 400 data 141,  62,   3, 141,  63,   3, 141,  64
IH 410 data   3, 169,   1,  76, 105, 249, 174,   0
ID 420 data  28, 202,  76, 113,   3, 174,   0,  28
EJ 430 data 232, 138,  41,   3, 141, 145,   3, 173
OD 440 data   0,  28,  41, 252,  13, 145,   3, 141
DM 450 data   0,  28, 160,   5, 162, 255, 202, 208
HJ 460 data 253, 136, 208, 250,  96, 234,  76,  38
FF 470 data   3,   0
IC 480 if ck<>14432 then print"bad data":stop
PN 490 print#15,"u2";2;0;t;s:rem write blk
BP 500 print"use this line to erase disk:"
IJ 510 print "[down]open15,8,15:open2,8,2,";
DH 520 print chr$(34)"#0"chr$(34)":print#";
HL 530 print "15,"chr$(34)"b-e"chr$(34);
HD 540 print "2;0;";t;";"s;":close15:close2"
MK 550 close15:close2:end:rem finished!
OG 560 rem*********************************
DG 570 rem *****      read disk name      *****
CI 580 rem*********************************
IO 590 open15,8,15,"i0":rem refresh bam
NA 600 rn$=chr$(144)+chr$(7)+chr$(16)
PB 610 print#15,"m-r"rn$:rem point to name
HK 620 fori=1to16:get#15,p$:n$=n$+p$:next
BJ 630 print"[2 down]disk name:";n$
KA 640 input"[down]write death sector";yn$
PF 650 if left$(yn$,1)<>"y" then end
KK 660 print"[down]where do you want death sector?"
KC 670 input"track ";t:input"sector ";s
EM 680 return
```

Listing 2: Examine the data in any sector on a disk, including those beyond the normal maximum of 35.

```
CK 100 rem*******************************
AK 110 rem***          read any sector          ***
AB 120 rem***          by frank digioia         ***
AM 130 rem*******************************
CK 140 rem       the transactor  vol 7 issue 6
EN 150 rem*******************************
EL 160 open15,8,15,"i0"
OC 170 t=1:s=1:x=0:dim a(255)
OM 180 ts$="m-w"+chr$(6)+chr$(0)+chr$(2)
LK 190 sj$="m-w"+chr$(0)+chr$(0)+chr$(1)
LN 200 input"[clr]track";t
CO 210 if t<1 or t>40 then 200
PE 220 input"sector";s:if s<0 then 220
PA 230 print#15,ts$;chr$(t)chr$(s)
CN 240 print#15,sj$;chr$(176):rem do seek
KL 250 gosub 420:rem await completion
NC 260 print#15,ts$;chr$(t)chr$(s)
OM 270 print#15,sj$;chr$(128):rem do read
IN 280 gosub 420:rem await completion
PB 290 print#15,"m-r"chr$(0)chr$(3)chr$(255)
BE 300 print"[clr]":fori=0to255:get#15,a$
BB 310 a(i)=asc(a$+chr$(0)):print a$;:next
IB 320 print"hit any key for screen codes"
EN 330 poke198,0:wait198,1:geta$:print"[clr][down]"
KD 340 fori=0to255:poke1024+i,a(i):next
II 350 print"[5 down]hit a key for ascii codes"
OO 360 poke198,0:wait198,1:geta$:print"[2 down]"
AN 370 fori=0to255:print a(i);:next
BI 380 print#15,"i0":close15:end
EM 390 rem*******************************
PA 400 rem          wait for activity complete
IN 410 rem*******************************
CG 420 fori=1to3000:next:rem initial delay
EN 430 print#15,"m-r"chr$(0)chr$(0)
DC 440 get#15,a$:a=asc(a$+chr$(0))
AE 450 if a=176 then print"seeking track"
BE 460 if a=128 then print"reading sector"
DD 470 x=x+1:if a>127 and x<11 then 420
DB 480 if a=1 then print"okay":x=0:return
NH 490 print"error!!!":print#15,"u:":stop
```

Listing 3: Assembler source code for the drive-resident program to destroy a disk. This code is written into the 1541's RAM by "Death Sector" in Listing 1.

```
KN 100 ;
HB 110 ;disk destroyer/src
OO 120 ;
JJ 130 ;frank digioia
HI 140 ;09/28/86
MA 150 ;
KA 160 *=$0300                  ;not relocatable!!!
AC 170 ;
CP 180 entry    nop
HI 190          nop
BJ 200          nop
CE 210          lda  #$01       ;track 1
GI 220          sta  $06        ;set track
NJ 230          lda  #$00       ;sector 0
PM 240          sta  $07        ;set sector
OK 250          lda  jmpins     ;copy jmp instr to
EE 260          ldx  jmpins+1   ;entry point. since
DC 270          ldy  jmpins+2   ;execution will jump
KC 280          sta  entry      ;to entry as soon as
LB 290          stx  entry+1    ;the $e0 is placed in
BP 300          sty  entry+2    ;the job queue.
CH 310          lda  #$e0       ;job code=execute
OF 320          sta  $00        ;place in job queue
PH 330 wait     lda  $00        ;give it a few mili-
FB 340          bmi  wait       ;seconds to kick in.
KE 350          rts
ON 360 ;
AP 370 purge    =    *
EI 380          lda  $22        ;get current track
HF 390          beq  *+6        ;zeroprintprintprint
EJ 400          cmp  #$24       ;is it bigger than 36
GJ 410          bcc  back       ;no/go ahead & move
KB 420 ;
JP 430          lda  #$2e       ;set track to 46
DD 440          sta  $22        ;that should fix it
ID 450 ;
CK 460 back     =    *          ;step head back to trak 1
IB 470          jsr  step       ;step head back 1/2 trak
CC 480          jsr  step       ;step head back 1/2 trak
HF 490          dec  $22        ;decr current track
OG 500          bne  back       ;until at track zero
II 510          inc  $22        ;set track to 1
OH 520 ;
NA 530 wipe     jsr  $fda3      ;wipe track
JO 540          jsr  $fe00      ;set read mode on
OC 550          lda  $22        ;get current track
BK 560          cmp  #36        ;bigger than 35print
FL 570          bcs  finish     ;done/no problem
LP 580          inc  $22        ;bump track
AM 590          jsr  stepb      ;move 1/2 track
KM 600          jsr  stepb      ;move 1/2 track
DE 610          jmp  wipe       ;wipe next track
CO 620 ;
OI 630 finish   =    *          ;cleanup and get out
JE 640          lda  nopins     ;get nop instr
IJ 650          sta  wipe       ;render code harmless
DD 660          sta  wipe+1
PD 670          sta  wipe+2
FC 680          lda  #$01       ;set return code
PG 690          jmp  $f969      ;main error routine
CD 700 ;
GA 710 step     =    *          ;step head back 1/2 trk
DI 720          ldx  $1c00      ;get control reg
PH 730          dex             ;rotate low order bits
KJ 740          jmp  stepb+4
EG 750 ;
AP 760 stepb    =    *          ;step head frwd 1/2 trk
FL 770          ldx  $1c00      ;get control reg
GN 780          inx             ;rotate low order bits
HG 790          txa             ;put in .a
EK 800          and  #$03       ;isolate low order bits
DG 810          sta  temp       ;save it
HC 820          lda  $1c00      ;get control reg again
EG 830          and  #$fc       ;clear low order bits
OP 840          ora  temp       ;set lower order bits
PE 850          sta  $1c00      ;set control reg
CN 860 ;
HJ 870          ldy  #$05       ;timing loop iterations
FH 880          ldx  #$ff       ;inner loop timing
GM 890 delay    dex             ;allow head to settle
JH 900          bne  delay      ;inner loop
JG 910          dey             ;outer loop
IF 920          bne  delay      ;continue
OI 930          rts
CC 940 ;
FN 950 nopins   nop             ;this is data
AN 960 jmpins   jmp  purge      ;this is data
JA 970 temp     .byte $00       ;working storage
```

Auto Transmission for the Commodore 64

Doug Resenbeck
Rockford, IL

This program developed out of necessity. Over a period of time I have collected programs that convert Basic programs to make them auto-run – when you LOAD them with ',8,1' they will automatically RUN upon completion of the LOAD. Some of these programs – Jim Butterfield's 'LockDisk' is one – also alter system vectors in such a way as to cause the program to re-RUN whenever READY mode occurs (due to an error, or the program ending).

This is a good feature in most cases, but not always. Suppose you write a nice program. You RUN it and test it and it seems to be all you intended it to be. So you LOAD up your favorite auto-run creator and soon your program has auto-run capability. Well, you no longer need the original Basic program, so you scratch it in favor of keeping only the auto-run version. If you're like me, sooner or later you'll want to make some changes to that program. Sorry, it's an auto-run program now. If you LOAD it ',8,1' it will start running on its own. Depending on built in safeguards, you may or may not be able to stop your program for editing purposes. If you LOAD it ',8' the program remains unlistable. This is the problem I encountered.

My first approach to a solution was to look through all of my Commodore related magazines for a program to reverse the auto-run process. No luck there. It seemed apparent I would have to tackle the problem from the ground level. I went through each of my auto-run creating programs, analyzing Basic listings and disassembling machine code. My efforts turned up similarities in ALL of the auto-run creating programs.

First, lower memory vectors are written to the disk, with changes that will later implement the auto-run feature. Next, all of the remaining memory between the vector table and the start of Basic, including screen memory, is written to the disk. Finally, the Basic program itself is also written. This explains why the program is unlistable when it is LOADed ',8': the lower memory vectors are at the start of Basic program space. It also explains how auto-run programs are implemented. When the program is loaded ',8,1', the program starts LOADing over the top of the vector table, making its own changes in the process. The original Basic program falls right back where it originally sat. Upon returning to READY mode, the altered vectors cause it to begin running.

Something else of importance: No matter where in memory the load begins, its address is recorded on the disk with the program. Comparing this address with the start of Basic at 2049 (Basic programs start loading at 2049) the difference will equal the number of bytes separating the beginning of an auto-run program with the original undoctored version.

Now to write a utility to do the job. First, INPUT the LOADing address of the auto-run program and subtract that from 2049. GET that many bytes with a FOR-NEXT loop. Next, create a new program on disk by first writing a CHR$(1) and a CHR$(8) (the new LOAD address in low byte/high byte format) then copying the Basic portion of the auto-run file one byte at a time until the whole file is transferred.

Now for the test. I wrote a one line program and saved it to disk. Then I used all of the auto-run creator programs I had on it. The test was to see if my program could convert each of these auto-run programs back into their original Basic format. The test was a success!

I hope all of you will find this utility as useful as I have.

C64 Auto Transmission:
Remove Auto-RUN from BASIC programs.

```
FA  10 rem * this program changes auto-run
BK  20 rem * programs back to their
EF  30 rem * original basic format
LI  40 rem * written by doug resenbeck 10/86
ND  50 printchr$(147)chr$(17)chr$(17)
DA  60 print" name of basic auto-run program"
JA  70 print:inputar$:t=len(ar$)
GA  80 gosub440
EL  90 ifs=1thens=0:goto50
PG 100 printchr$(147)chr$(17)chr$(17)
PF 110 print" new name with auto-run removed"
PB 120 print:inputua$:t=len(ua$)
ID 130 gosub440
GD 140 ifs=1thens=0:goto100
PF 150 printchr$(147)chr$(17)"auto-run    "ar$
BM 160 printchr$(17)"to"
EE 170 printchr$(17)"un/auto-run  "ua$
FH 180 printchr$(17)"please wait"
BM 190 open2,8,2,ar$+",p,r"
KP 220 open8,8,8,ua$+",p,w"
BM 250 get#2,a$,b$
DC 260 ifa$=""thena$=chr$(0)
IH 270 a=asc(a$)
LD 280 ifb$=""thenb$=chr$(0)
LK 290 b=asc(b$):c=a+256*b:d=2049-c
BO 300 ifd>0then320
BE 310 printchr$(147)chr$(17)"program is not a
       basic auto-run program":goto380
LK 320 forx=1tod:get#2,a$:next
LF 330 print#8,chr$(1);
KH 340 print#8,chr$(8);
BM 350 get#2,a$:sw=st:ifa$=""thena$=chr$(0)
LC 360 print#8,a$;:ifsw=0then350
EL 370 printchr$(147)chr$(17)"done"
KE 380 close2:close8
ME 390 gosub480
GE 400 ife>0then500
DH 410 printchr$(17):input"more";n$
OF 420 ifn$="y"then50
OK 430 end
CG 440 ift<17thenreturn
IC 450 printchr$(17):print"filename too long"
OH 460 forx=1to1500:next:s=1
CP 470 return
PH 480 close15:open15,8,15
IO 490 input#15,e,e$,tr,s:close15:return
HN 500 print"disk error #";e
PI 510 printe$
FJ 520 print"track";tr
MI 530 print"sector";s
KO 540 close2:close8
```

Common Code

Jack R. Farrah
Cincinnati, Ohio

. . .*It sounded like a job for* THE COMPUTER!

The Scenario:

That nifty machine language program you've labored over for weeks is finally complete. A natural for submission to The Transactor, you think. But wait! Your programming technique (perhaps like mine) is a bit long winded. Written and debugged in sections that were thought out independently, it works but it's extremely long. What if it's too long? Would it have a better chance if it were 100, 200, 500 bytes shorter? Frantically you scan the code, looking for things to eliminate, shorten or combine. No good! Your eyes keep glazing over at the task. After all, you've been at this for weeks and the thought of a major re-write is not pleasant.

This is not fiction friend. It happened to me. After all the searching, all the cutting out of title screens, help screens and superfluous user prompts, I still had over 3500 bytes of code. In desperation my mind turned to subroutines. Maybe, amongst these thirty pages of assembly, there were identical sections of code that I could consolidate into subroutines to shorten the program. But how to find them? It sounded like a job for THE COMPUTER!

The Solution:

A typical, knee-jerk reaction from a programming buff you say? Well, perhaps, but there's nothing that pumps me up more to sit down and write a program than to perceive a tangible need. More so if its my own! Common Code is my answer to this need. Here's what it does. After loading and running, the program asks you for a starting address in memory, in hexadecimal (do not include $'s). Next, an ending address. Enter <RETURN> after each input. Then you are asked whether you want the program output to go to the screen or your printer. Enter "s" or "p". If you are specifying a printer, it must be turned on before hitting the "p". Finally, you are given the opportunity to specify how many bytes of identical code you want the program to find. Any number between 00 and FF (2 and 255) is acceptable. An 00 or 01 entry will exit you from the program. The default setting is 7 bytes and is displayed at the prompt. Enter <RETURN> only to accept the default.

From here on the program begins its search and display routines. Each program byte is taken in turn as the beginning of a potential subroutine. It is checked against the following bytes in the program. If a full match is found, the start address is printed, then each matching group's start address is printed indented below it. There's no requirement that you search only between the "real" start and end of your program. You can limit the search to those areas where you want to concentrate. Assuming your program has text and or data tables at the beginning or end, enter addresses that omit those areas of code.

The checking routines run fast and depending on the program length, the byte length being checked and the number of matches found, the output to your screen may flash by too quickly. You can pause the listing (and the search) by hitting the space bar. Restart by hitting it again. Be aware that the space bar is only checked during the printout routine.

When complete, the default byte count is reinstated, the printer file closed and you are returned to BASIC with a READY prompt. All that's left is for you to check the code from an assembled program printout, against the matching groups. Pick the viable subroutines and revise your code.

The Works:

Common Code is written to reside in the normal BASIC portion of memory so that is has the least possibility of memory conflict with the machine language programs you will check with it. The functioning of the program is relatively straight forward. There are four main addresses tracked and or updated during the course of execution. The Start Address you enter is the beginning byte to try and match. The Check Address is the first byte where a match is attempted. Check is located initially at Start Address + N bytes. N is the value you've specified for the byte length match. Any address closer to Start would not allow the requisite length in the first group. If a match is found at Check, N-1 more bytes are checked. If they match, we print the addresses. Whether or not they match, Check is incremented by 1 and we start again. After the entire program has been checked against Start, Start is incremented by 1 and the whole thing happens again! You'll notice the output speed up as we move through the program, because the farther Start moves towards the end, the shorter the length of bytes its must check becomes.

Two addresses at the end are used to determine when one pass or the whole process is complete. End Check is your program's ending address minus (N-2). This is the address from which an inadequate number of bytes remain to become a N length subroutine. Each time Check is incremented it is compared with this address. If they match, its time to raise the Start Address by one, re-formulate Check and begin again. Match Check is N bytes back into the program from End Check. Each time Start is incremented it is compared with Match. If they are the same, there are an insufficient number of bytes remaining in the program to get two matching groups. At this point we're done.

The Caveats:

Common Code will provide you with all the information you need to identify existing subroutine material within your program, but:

1. If you've written different code in two places in your program that performs the same function, Common Code won't find it (Obvious, yes, but I thought I'd mention it.).
2. Because the program leaves no stone unturned in searching for matching groups, more stones than you need are the result. Not all matches will occur on valid boundaries. A match may begin with an operand rather than an opcode. If you are searching for a small byte string length and the program contains common sections of greater length, output will show up as multiple groups, each separated by one byte.
3. Some groups will contain branches or jumps outside of themselves and will therefore be unusable.
4. Finally, some usable groupings may, to the reader of your source, make the program intent less clear. Common Code will provide you with opportunities. You must decide where clarity takes precedence over brevity.

How did Common Code perform on my long program? I was able to achieve a 10% reduction in program length (around 350 bytes). Don't take this as a typical result in applying this utility. Your results depend entirely on what's in your program. Good hunting!

Common Code: Creates ML module on disk

```
IA 1000 rem save"0:common code.gen",8
KJ 1010 rem a locate & display utility to
CG 1020 rem find identical code sequences
IJ 1030 rem in machine language programs
EI 1040 :
MO 1050 rem this program will create
BC 1060 rem a load and run module on
GK 1070 rem disk called 'common code'
GF 1080 rem
LG 1090 for j=1 to 910 : read x
BO 1100 ch=ch+x : next
KD 1110 if ch<>98156 then print"checksum error"
        : end
FB 1120 print "data ok, now creating file": print
CI 1130 restore
DO 1140 open8,8,8,"0:common code,p,w"
CP 1150 print#8,chr$(1)chr$(8);
BL 1160 for j=1 to 910 : read x
CB 1170 print#8,chr$(x); : next
EI 1180 close 8
FO 1190 print "prg file 'common code' created. . .
KK 1200 print "this generator no longer needed.
IN 1210 rem
IK 1220 data 11,  8, 10,  0,158, 50, 48, 54
OO 1230 data 49,  0,  0,  0,169,147, 32,210
HC 1240 data 255,162,  0,142,138, 11,142,140
DA 1250 data 11,189, 35, 10,240,  6, 32,210
HP 1260 data 255,232,208,245,162,  0, 32, 21
IB 1270 data 11, 32,165, 10,224,  5,176, 13
BF 1280 data 202,224,255,240, 29,189,130, 11
FB 1290 data 32,191, 10,144,243,169, 13, 32
JA 1300 data 210,255, 32, 26, 11,162,  0,189
DI 1310 data 76, 10,240,197, 32,210,255,232
OB 1320 data 208,245,162,  0, 32, 26, 11,189
KB 1330 data 130, 11, 32,179, 10, 10, 10, 10
KJ 1340 data 10,157,130, 11,232,189,130, 11
MG 1350 data 32,179, 10,224,  3,240, 10, 24
GD 1360 data 109,130, 11,141,130, 11,232,208
ML 1370 data 222, 24,109,132, 11,141,132, 11
GN 1380 data 173,138, 11,208, 42,173,130, 11
GI 1390 data 133,252,173,132, 11,133,251,169
EL 1400 data 13, 32,210,255,141,138, 11,162
CP 1410 data 0,189, 89, 10,240,  6, 32,210
HJ 1420 data 255,232,208,245,162,  0, 32, 21
HC 1430 data 11, 32,165, 10, 76, 45,  8,173
OP 1440 data 130, 11,141,137, 11,173,132, 11
KC 1450 data 141,136, 11,169, 13, 32,210,255
DL 1460 data 162,  0,189,110, 10,240,  6, 32
MM 1470 data 210,255,232,208,245, 32,228,255
PO 1480 data 240,251,201, 80,240,  6,201, 83
PM 1490 data 208,243,240,  3, 32,214, 10,162
FN 1500 data 0,189,143, 10,240,  9, 32,210
EP 1510 data 255,232,208,245, 76, 62,  8, 32
CD 1520 data 21, 11, 32,228,255,240,251,201
CB 1530 data 13,240, 46, 32,210,255, 32,191
CE 1540 data 10, 32,179, 10, 10, 10, 10, 10
NP 1550 data 141,141, 11, 32,228,255,240,251
GD 1560 data 32,210,255, 32,191, 10, 32,179
GN 1570 data 10, 24,109,141, 11,201,  2,144
FK 1580 data 203,141,139, 11,169, 13, 32,210
AB 1590 data 255, 32, 26, 11,169, 13, 32,210
FG 1600 data 255,173,139, 11, 56,233,  2,141
OM 1610 data 141, 11,173,136, 11, 56,237,141
NL 1620 data 11,141,136, 11,144, 35,173,136
PE 1630 data 11, 56,237,139, 11,141,134, 11
PP 1640 data 144, 29,173,137, 11,141,135, 11
ML 1650 data 165,251, 24,109,139, 11,133,253
DN 1660 data 165,252,105,  0,133,254, 76,120
EC 1670 data 9,206,137, 11, 76, 71,  9,173
IE 1680 data 137, 11,233,  1, 76, 86,  9,160
PH 1690 data 0,177,251,209,253,240, 65, 24
DK 1700 data 165,253,105,  1,133,253,165,254
HI 1710 data 105,  0,133,254,165,253,205,136
GD 1720 data 11,208,228,165,254,205,137, 11
DF 1730 data 208,221, 24,165,251,105,  1,133
GM 1740 data 251,165,252,105,  0,133,252,162
LP 1750 data 0,142,140, 11,165,251,205,134
DD 1760 data 11,208, 10,165,252,205,135, 11
JC 1770 data 208,  3, 76,118, 11, 76, 89,  9
HL 1780 data 162,  0,160,  0,232,236,139, 11
CB 1790 data 240, 10,200,177,251,209,253,240
GE 1800 data 243, 76,128,  9,173,140, 11,240
BP 1810 data 37, 32,237, 10,169, 32, 32,210
MM 1820 data 255, 32,210,255,169, 36, 32,210
CF 1830 data 255,160,  0,192,  2,240,  9,185
MG 1840 data 253,  0,153,141, 11,200,208,243
```

```
BO 1850 data 32, 94, 11, 76,128,  9,169, 13
IO 1860 data 32,210,255,169, 36, 32,210,255
CG 1870 data 160,  0,192,  2,240,  9,185,251
BC 1880 data  0,153,141, 11,200,208,243, 32
PJ 1890 data 94, 11,169,  1,141,140, 11, 76
MK 1900 data 218,  9, 32, 32, 32, 18, 67, 79
AB 1910 data 77, 77, 79, 78, 32, 67, 79, 68
GG 1920 data 69,146, 13, 13, 83, 84, 65, 82
PA 1930 data 84, 32, 65, 68, 68, 82, 69, 83
IP 1940 data 83, 32, 73, 78, 32, 72, 69, 88
IM 1950 data 32, 13,  0, 73, 78, 80, 85, 84
CN 1960 data 32, 69, 82, 82, 79, 82, 13,  0
CF 1970 data 69, 78, 68, 32, 65, 68, 68, 82
BD 1980 data 69, 83, 83, 32, 73, 78, 32, 72
CA 1990 data 69, 88, 32, 13,  0, 79, 85, 84
HC 2000 data 80, 85, 84, 32, 84, 79, 32, 18
LN 2010 data 83,146, 67, 82, 69, 69, 78, 32
NK 2020 data 79, 82, 32, 18, 80,146, 82, 73
IF 2030 data 78, 84, 69, 82, 13,  0, 66, 89
HJ 2040 data 84, 69, 32, 76, 69, 78, 71, 84
PF 2050 data 72, 32, 73, 78, 32, 72, 69, 88
CB 2060 data 13, 55,157,  0, 32,207,255,201
AB 2070 data 13,240,  6,157,130, 11,232,208
FF 2080 data 243, 96,201, 58,176,  4, 56,233
LJ 2090 data 48, 96, 56,233, 55, 96,201, 71
EE 2100 data 176, 14,201, 65,176,  8,201, 58
NL 2110 data 176,  6,201, 48,144,  2, 24, 96
LH 2120 data 104,104, 76, 62,  8,169,  7,162
OA 2130 data  4,160,255, 32,186,255,169,  0
GK 2140 data 32,189,255, 32,192,255,162,  7
HK 2150 data 32,201,255, 96,169,  0,141,138
FN 2160 data 11,165,203,201, 64,240, 23,201
CK 2170 data 60,208, 19,173,138, 11,208, 19
FE 2180 data 165,203,201, 64,208,250,169,  1
DP 2190 data 141,138, 11, 76,242, 10,173,138
HM 2200 data 11,208,222, 96,169,  0,133,204
AB 2210 data 96,169,  1,133,204, 96,162,  1
AJ 2220 data 160,  0,189,141, 11, 41,240, 74
LP 2230 data 74, 74, 74,201, 10,176, 26, 24
BM 2240 data 105, 48,153,130, 11,200,192,  3
NK 2250 data 240, 21,176, 32,224,  0,240, 21
NM 2260 data 189,141, 11, 41, 15,202, 76, 44
EA 2270 data 11, 24,105, 55, 76, 51, 11,189
AA 2280 data 141, 11, 76, 68, 11,160,  2,162
GM 2290 data  0, 76, 35, 11, 96, 32, 31, 11
BB 2300 data 162,  0,224,  4,240,  9,189,130
NO 2310 data 11, 32,210,255,232,208,243,169
MA 2320 data 13, 32,210,255, 96, 32,204,255
PE 2330 data 169,  7,141,139, 11, 32,195,255
JK 2340 data 96,  0,  0,  0,  0,  0,  0,  0
JE 2350 data  0,  0,  7,  0,  0,  0
```

Common Code: PAL source code

```
DA 1000 rem save"0:common code.pal",8
NB 1010 open 8,8,8,"0:common code,p,w
NM 1020 sys700
JC 1030 .opt o8
PJ 1040 ; * common code by jack r. farrah
MH 1050 ;program to find identical code sequences
MN 1060 ;in machine language programs for use
CN 1070 ;as possible subroutines.
CP 1080 ;tested program to be in memory.
CG 1090 ;all user inputs in hex.
PF 1100 ;screen or printer output.
DP 1110 ;space bar pauses listing.
IF 1120 ; * constants *
NI 1130 chrin    =  $ffcf      ;get mult. char.input
NC 1140 chrout   =  $ffd2      ;print to device
KO 1150 getin    =  $ffe4      ;get single char.
EE 1160 stadd    =  $fb        ;start address
CI 1170 ckadd    =  $fd        ;check address
GG 1180 setlfs   =  $ffba      ;set log. file
PL 1190 setnam   =  $ffbd      ;name file
ED 1200 open     =  $ffc0      ;open file
OE 1210 close    =  $ffc3      ;close file
EF 1220 chkout   =  $ffc9      ;set output file
CG 1230 clrchn   =  $ffcc      ;restore defaults
CA 1240 * = $0801              ;2049
LK 1250 .word twobrk           ;forward pointer
HB 1260 .byte 10,0             ;line number
DD 1270 .byte $9e              ;"sys" keyword token
DH 1280 .asc "2061"            ;sys address
AL 1290         brk
PL 1300 twobrk  .word 0
JA 1310         lda  #147      ;clear screen
PP 1320         jsr  chrout
IK 1330 begin   ldx  #0
PJ 1340         stx  inflg     ;clear flags
JN 1350         stx  mtchflg
FO 1360 ;*get user start address*
HF 1370 titl    lda  title,x   ;print prog. name
NI 1380         beq  start     ;and start add.
EM 1390         jsr  chrout    ;input message
EF 1400         inx
GB 1410         bne  titl
MD 1420 start   ldx  #0        ;set index
JK 1430         jsr  cron      ;blink cursor
CE 1440         jsr  get       ;get address
EA 1450 check   cpx  #5        ;">4 characters?
IN 1460         bcs  error     ;only want 4
NL 1470 ck1     dex            ;reset for cr counted
EN 1480         cpx  #255      ;only after 4
KL 1490         beq  convert   ;make binary
CD 1500         lda  hxadd,x   ;get hex ascii
OA 1510         jsr  eval      ;check if valid
GC 1520         bcc  ck1       ;ok.get next char.
OI 1530 ;*error message loop*
GK 1540 error   lda  #$0d      ;cr
FO 1550         jsr  chrout
IL 1560         jsr  crof      ;turn off cursor
JG 1570         ldx  #0
FE 1580 er1     lda  ermess,x  ;print error message
CJ 1590         beq  begin     ;start over
HB 1600         jsr  chrout
GC 1610         inx
CF 1620         bne  er1
JI 1630 ;*change ascii hex to binary & store
IE 1640 convert ldx  #0        ;set index
NN 1650         jsr  crof      ;unblink cursor
MH 1660 loop    lda  hxadd,x   ;get ascii
GK 1670         jsr  makbi     ;make binary
EG 1680         asl            ;shift value into
LJ 1690         asl            ;high nybble position
PE 1700 asl
JF 1710 asl
LN 1720         sta  hxadd,x   ;save it
KC 1730         inx            ;raise index
DP 1740         lda  hxadd,x   ;get next ascii
GP 1750         jsr  makbi     ;make binary
ND 1760         cpx  #3        ;"4th character?
HA 1770         beq  over      ;yes. finish here
OC 1780         clc            ;no
LB 1790         adc  hxadd     ;add to high nybble
FG 1800         sta  hxadd     ;save combined value
KH 1810         inx            ;raise index
IL 1820         bne  loop      ;always branch
KO 1830 over    clc            ;add low nybble of
FJ 1840         adc  hxadd+2   ;low byte to high
HM 1850         sta  hxadd+2   ;and save it
ND 1860 end     lda  inflg     ;"done end address?
AI 1870         bne  output    ;yes.flag set
FE 1880         lda  hxadd     ;no.save start add.
```

```
MF 1890          sta  stadd+1    ;on zero page
NM 1900          lda  hxadd+2
OC 1910          sta  stadd
KK 1920          lda  #$0d       ;cr
BG 1930          jsr  chrout
KP 1940          sta  inflg      ;set flag
JF 1950 ;*get user end address*
KK 1960          ldx  #0         ;clear index
JA 1970 end1     lda  endmess,x  ;print message
FN 1980          beq  next
NJ 1990          jsr  chrout
MK 2000          inx
OD 2010          bne  end1
PI 2020 next     ldx  #0         ;clear for char. count
BA 2030          jsr  cron       ;blink cursor
CB 2040          jsr  get        ;get the address
DO 2050          jmp  check      ;check &make binary
DB 2060 output   lda  hxadd      ;get binary end add.
MG 2070          sta  enck+1     ;and store in zero page
BI 2080          lda  hxadd+2
GK 2090          sta  enck
OF 2100          lda  #$0d       ;cr
FB 2110          jsr  chrout
HD 2120 ;*get output destination from user*
JJ 2130          ldx  #0
FC 2140 out1     lda  outmess,x  ;print message
MC 2150          beq  getit
HE 2160          jsr  chrout
GF 2170          inx
PL 2180          bne  out1
NL 2190 getit    jsr  getin      ;get 's' or 'p'
MC 2200          beq  getit      ;wait for key
NN 2210          cmp  #80        ;'p?
KE 2220          beq  print      ;yes. open printer
PK 2230          cmp  #83        ;'no. s?
CA 2240          bne  getit      ;no.go back for key
BO 2250          beq  byte       ;screen output
PF 2260 print    jsr  prout      ;open printer file
AH 2270 ;*get byte lgth. from user*
DF 2280 byte     ldx  #0
LI 2290 bytlup   lda  bytmess,x  ;print message
AA 2300          beq  gtbyt
NN 2310          jsr  chrout
MO 2320          inx
BL 2330          bne  bytlup
ON 2340 erjmp    jmp  error      ;out of range avoider
NM 2350 gtbyt    jsr  cron       ;blink cursor
OO 2360 gt2      jsr  getin      ;get key
MC 2370          beq  gt2        ;wait for key
DN 2380          cmp  #$0d       ;'cr?
BA 2390          beq  setend     ;default selected
NI 2400          jsr  chrout     ;new value. print it
BL 2410          jsr  eval       ;check range
EJ 2420          jsr  makbi      ;make binary
KE 2430          asl             ;shift to hi nybble
DD 2440 asl
ND 2450 asl
HE 2460 asl
OM 2470          sta  hldr       ;save it
AM 2480 gt1      jsr  getin      ;get second char.
OH 2490          beq  gt1        ;wait for it
HI 2500          jsr  chrout     ;print iit
FB 2510          jsr  eval       ;check range
IP 2520          jsr  makbi      ;make binary
AF 2530          clc             ;add to hi nybble
KD 2540          adc  hldr
AM 2550          cmp  #2         ;'>1?
FB 2560          bcc  erjmp      ;<2 not allowed
LL 2570          sta  ckbyt      ;store new value
OD 2580          lda  #$0d       ;cr
FP 2590          jsr  chrout
AH 2600 ;*calculate end addresses*
CO 2610 setend   jsr  crof       ;unblink cursor
GG 2620          lda  #$0d       ;cr
NB 2630          jsr  chrout
GA 2640          lda  ckbyt      ;get lgth. to check
LN 2650          sec
IA 2660          sbc  #2         ;subtract 2
KC 2670          sta  hldr       ;temporary save
CO 2680          lda  enck       ;low byte end add.
DA 2690          sec
IJ 2700          sbc  hldr       ;subtract value
OE 2710          sta  enck       ;save new value
CF 2720          bcc  subhi      ;reduce hi byte
OI 2730 set1     lda  enck       ;get new end add.
FD 2740          sec
LA 2750          sbc  ckbyt      ;subtract byte lgth
FJ 2760          sta  mtchck     ;save as check value
LK 2770          bcc  sub2       ;reduce hi byte
DC 2780          lda  enck+1     ;get hi byte new end
DN 2790 set2     sta  mtchck+1   ;make same here
BJ 2800 set3     lda  stadd      ;start add. low byte
AH 2810          clc
LG 2820          adc  ckbyt      ;add byte lgth
PL 2830          sta  ckadd      ;check pointer
HE 2840          lda  stadd+1    ;hi byte
OE 2850          adc  #0         ;add carry
HE 2860          sta  ckadd+1    ;put in pointer
NI 2870          jmp  main       ;start main loop
MJ 2880 subhi    dec  enck+1
CK 2890          jmp  set1
NH 2900 sub2     lda  enck+1
LF 2910          sbc  #1
EM 2920          jmp  set2
MJ 2930 ;*main progam loop*
HN 2940 main     ldy  #0         ;clear for ind.add.mode
FM 2950          lda  (stadd),y  ;get value at start
FK 2960          cmp  (ckadd),y  ;next to check
DF 2970          beq  gotmtch    ;they match.check more.
ON 2980 ma1      clc             ;no match
BA 2990          lda  ckadd      ;add 1 to check add.
HJ 3000          adc  #1
GN 3010          sta  ckadd      ;store back
OH 3020          lda  ckadd+1    ;fix high byte
DL 3030          adc  #0
OO 3040          sta  ckadd+1    ;store
GN 3050          lda  ckadd      ;have we reached
PA 3060          cmp  enck       ;'end of possible bytes?
IE 3070          bne  main       ;no.start next series
BP 3080          lda  ckadd+1    ;lo bytes matched
HK 3090          cmp  enck+1     ;'hi bytes same?
LK 3100          bne  main       ;no.continue
IB 3110          clc             ;done with this series
NF 3120          lda  stadd      ;move start pointer
CI 3130          adc  #1         ;to next highest byte
CA 3140          sta  stadd      ;store it
IA 3150          lda  stadd+1    ;fix hi byte
FD 3160          adc  #0
BA 3170          sta  stadd+1
FP 3180          ldx  #0         ;clear flag to show print
JD 3190          stx  mtchflg    ;routine this is new add.
FM 3200          lda  stadd      ;compare start add.
LG 3210          cmp  mtchck     ;with last checkable byte
EE 3220          bne  return     ;no match low byte
HC 3230          lda  stadd+1    ;check hi byte
OM 3240          cmp  mtchck+1
NH 3250          bne  return     ;no match
HI 3260          jmp  exit       ;all done, close up
HI 3270 return   jmp  set3       ;out of range avoider
JC 3280 ;*check remaining bytes for match*
GF 3290 gotmtch  ldx  #0         ;clear indices
PC 3300          ldy  #0
FE 3310 lup      inx             ;x counts bytes matched
OF 3320          cpx  ckbyt      ;'checked all?
FA 3330          beq  prnt       ;yes.print 'em
LC 3340          iny             ;no.index to next byte
CG 3350          lda  (stadd),y  ;get next from start
HO 3360          cmp  (ckadd),y  ;check for equality
BB 3370          beq  lup        ;matches.get another
PA 3380          jmp  ma1        ;no match.move up a byte
EB 3390 ;*here if all bytes match*
LJ 3400 prnt     lda  mtchflg    ;'printed this stadd?
OM 3410          beq  prst       ;no, so print it
IA 3420 prnt1    jsr  wait       ;check for space bar
IH 3430          lda  #32        ;indent 2 spaces
HE 3440          jsr  chrout
BF 3450          jsr  chrout
HC 3460          lda  #36        ;$
FG 3470          jsr  chrout
MK 3480          ldy  #0         ;set upto get 2 bytes
GK 3490 mr2      cpy  #2
HM 3500          beq  mr1
ON 3510          lda  ckadd,y    ;get add. of matching bytes
CO 3520          sta  hldr,y     ;store for conversion
NF 3530          iny             ;get 2nd byte
OH 3540          bne  mr2        ;always branch
```

```
HL 3550 mr1     jsr  prnthx     ;convert and print add.
PF 3560         jmp  ma1        ;reset ckadd and loop again
OM 3570 ;*print start address matched*
MF 3580 prst    lda  #$0d       ;cr
NN 3590         jsr  chrout
DL 3600         lda  #36        ;$
BP 3610         jsr  chrout
FM 3620         ldy  #0         ;set to get 2 bytes
ID 3630 pr2     cpy  #2
GF 3640         beq  pr1
LF 3650         lda  stadd,y    ;get 1st byte
OG 3660         sta  hldr,y     ;save for conversion
DM 3670         iny             ;set for next byte
MG 3680         bne  pr2
GN 3690 pr1     jsr  prnthx     ;convert and print
PD 3700         lda  #1         ;set flag to show
PF 3710         sta  mtchflg    ;stadd was printed
NM 3720         jmp  prnt1      ;go print ckadd
NP 3730 ;*text*
OL 3740 title   .byte $20,$20,$20,$12
JB 3750 .asc "common code" : .byte $92,$0d,$0d
KK 3760 .asc "start address in hex ": .byte $0d,$00
AE 3770 ermess  .asc "input error": .byte $0d,$00
GN 3780 endmess.asc "end address in hex ": .byte $0d,$00
FO 3790 outmess .asc "output to ": .byte $12
LD 3800 .asc "s": .byte $92
ME 3810 .asc "creen or ": .byte $12
ME 3820 .asc "p": .byte $92
PH 3830 .asc "rinter" : .byte $0d,$00
GG 3840 bytmess .asc "byte length in hex": .byte $0d,$37,$9d,$00
KN 3850 ;*subroutines*
JC 3860 get     jsr  chrin      ;get user input
FK 3870         cmp #$0d        ;"cr?
HI 3880         beq  done       ;yes.exit routine
PI 3890         sta  hxadd,x    ;store ascii char.
GC 3900         inx             ;raise idex for next
FL 3910         bne  get        ;go get it
CJ 3920 done    rts
NF 3930 ;*make 1 byte ascii in a binary*
IL 3940 makbi   cmp #58         ;"=>9?
AK 3950         bcs  let        ;yes, its a letter
AD 3960         sec             ;no so subtract 48
ND 3970         sbc  #48        ;for equiv. number
PM 3980         rts             ;return
DC 3990 let     sec             ;for a to f
PM 4000         sbc  #55        ;subtract 55
GJ 4010         rts
CN 4020 ;*check if valid hex ascii*
PF 4030 eval    cmp #71         ;"=>g?
GD 4040         bcs  bad        ;yes, no good
PF 4050         cmp #65         ;"its < g.is it =>a?
BP 4060         bcs  good       ;yes, its valid
AG 4070         cmp #58         ;"its <a.is it =>:?
GG 4080         bcs  bad        ;yes.no good
JD 4090         cmp #48         ;"<:.is it <0?
KD 4100         bcc  bad        ;yes. no good
HI 4110 good    clc             ;range ok.
NO 4120         rts             ;back to caller
KN 4130 bad     pla             ;invalid.pull return
FA 4140         pla             ;add. from stack
PK 4150         jmp  error      ;user restart
DM 4160 ;*set up printer file*
IB 4170 prout   lda  #7         ;file #
BI 4180         ldx  #4         ;device
BI 4190         ldy  #$ff       ;bogus second. add.
NN 4200         jsr  setlfs     ;define the file
II 4210         lda  #00        ;no name, no length
BL 4220         jsr  setnam     ;required call
BP 4230         jsr  open       ;open file 7
GH 4240         ldx  #7         ;set file 7 for output
MF 4250         jsr  chkout
JH 4260         rts             ;back to caller
JH 4270 ;*check/accept space bar pause*
MO 4280 wait    lda  #0         ;clear flag to show
JI 4290         sta  inflg      ;we're not waiting
JH 4300 wa2     lda  $cb        ;current key pressed
OJ 4310         cmp #64         ;64 = no key
EA 4320         beq  goon       ;no key, nothing to do
FM 4330         cmp #60         ;"space bar?
OO 4340         bne  goon       ;no, so ignore
FE 4350         lda  inflg      ;was space bar.
IM 4360         bne  g1         ;if set,wait is over
JE 4370 wa1     lda  $cb        ;start the wait
LF 4380         cmp #64         ;for space bar release
OA 4390         bne  wa1        ;keep waiting
LP 4400         lda  #1         ;set flag to show
GF 4410         sta  inflg      ;we're looking for 2nd
FD 4420         jmp  wa2        ;hit of space bar
KE 4430 goon    lda  inflg      ;if flag set
GN 4440         bne  wa2        ;keep looking
MK 4450 g1      rts             ;the waits over
BK 4460 ;*start cursor blink*
AL 4470 cron    lda  #0         ;clear this byte
FI 4480         sta  $cc        ;to start blink
GH 4490         rts
BK 4500 ;*stop cursor blink*
HG 4510 crof    lda  #1         ;set byte to
NK 4520         sta  $cc        ;stop blink
OJ 4530         rts
EN 4540 ;*2 byte binary to 4 byte ascii hex*
DG 4550 makhx   ldx  #1         ;x set to get byte
IL 4560         ldy  #0         ;y set to save ascii
AA 4570 hx3     lda  hldr,x     ;get byte(hi first)
HM 4580         and  #$f0       ;mask low nybble
KH 4590         lsr             ;shift hi nybble to low
BN 4600 lsr
LN 4610 lsr
FO 4620 lsr
AO 4630 hx1     cmp #10         ;"=>10?
NJ 4640         bcs  admor      ;yes, make letter
MO 4650         clc             ;no.number
LA 4660         adc  #48        ;add 48 for ascii
MJ 4670 hx2     sta  hxadd,y    ;store it
DC 4680         iny             ;raise counter
EO 4690         cpy  #3         ;"done 3 nybbles?
EF 4700         beq  skip       ;yes,do 4th
BA 4710         bcs  dun        ;y>3.we're done
DK 4720         cpx  #0         ;"y<3.hibyte done?
ON 4730         beq  nxtbyt     ;yes.do low
AN 4740         lda  hldr,x     ;no.get lo nyb,hi byte
HE 4750 hx4     and  #$0f       ;mask hi nybble
MK 4760         dex             ;lower counter
LM 4770         jmp  hx1        ;make ascii
IM 4780 admor   clc             ;convert binary letter
LE 4790         adc  #55        ;to ascii by
NP 4800         jmp  hx2        ;adding 55
OP 4810 skip    lda  hldr,x     ;get lo byte last time
FL 4820         jmp  hx4        ;do lo nybble
CJ 4830 nxtbyt  ldy  #2         ;reset indices fo
LM 4840         ldx  #0         ;2nd address byte
FE 4850         jmp  hx3        ;loop again
IC 4860 dun     rts             ;return
NO 4870 ;*print hex add.stored in hxadd*
KG 4880 prnthx  jsr  makhx      ;binary to hex
MB 4890         ldx  #0         ;clear index
CJ 4900 lupe    cpx  #4         ;do 4 numbers
HI 4910         beq  fin
BA 4920         lda  hxadd,x    ;get ascii hex
PB 4930         jsr  chrout     ;print it
EM 4940         inx             ;point to next char.
FN 4950         bne  lupe       ;always branch
FE 4960 fin     lda  #$0d       ;cr
BE 4970         jsr  chrout
HL 4980         rts             ;return
GD 4990 ;*program finished, clean up*
CA 5000 exit    jsr  clrchn     ;reset default devices
MH 5010         lda  #7         ;default value & file#
DE 5020         sta  ckbyt      ;save it
FB 5030         jsr  close      ;close file 7
FD 5040         rts             ;back to basic
EE 5050 ;*storage*
AD 5060 hxadd   .byte 0,0,0,0   ;4 bytes to hold ascii hex
CD 5070 mtchck  .byte 0,0       ;last add. to check
MM 5080 enck    .byte 0,0       ;last add. for match
NB 5090 inflg   .byte 0         ;user add. input flag
FG 5100 ckbyt   .byte $07       ;# bytes to match
OI 5110 mtchflg .byte 0         ;new group flag
LH 5120 hldr    .byte 0,0       ;temporary storage
GO 5130 .end
```

Getting Around With Gogo Dancer

Chris Miller
Kitchener, Ontario

One of the nicest things Commodore did for the 64 was build in a fairly gutless Basic with lots of RAM vectors for hackers to play with. I'm aware that the world probably does not really need another CHRGET wedge but when this idea came to me, I couldn't seem get it out of my mind. As a matter of fact, I almost named the program GOGO CRAZY, and now I would like to share it with you.

My favourite language will probably always be assembler because of the power it gives you over the flow of a program. GOGO DANCER WEDGE will allow Basic programmers to prance about their code with much the same agility.

GOTO and GOSUB now support three new types of parameters:

Expressions Of Your Desire

1. Yes, now instead of typing GOTO 1000 you could type GOTO 10*10*10 or GOTO 500+500. Or how about instead of ON X GOSUB 100,200,300 using GOSUB X*100. That's right, you can use any type of expression including variables. And you thought your programs were hard to understand now.

What's In A Name

2. Labels! Even better, you'll be able call your routines by name if you want to. This means you can do a dumb renumber or move stuff around without messing things up for your GOTOs and GOSUBs.

Macros (Kind Of)

3. Lastly, you can use string variables instead of literal label arguments. Imagine reading in your subroutine calls from DATA statements. Want to change the order in which things are done? Just change the DATA.

A Word Of Warning

These abilities can cut two ways. You can use GOGO DANCER to make your programs more maintainable and easily understood, or you can use it to confuse the living daylights out of anyone who lays eyes on them (including yourself).

Here are a few trivial examples to demonstrate the syntax:

```
  5 rem using expressions
 10 for ln = 100 to 500 step 100
 20 gosub ln
 30 next
 40 end
100 print "this";: return
200 print "could";: return
300 print "make";: return
400 print "you";: return
500 print "crazy";: return
```

```
  5 rem using labels
 10 gosub @hello
 20 gosub @goodbye
 30 end
 40 @hello
 50 print "hi there"
 60 return
 70 @goodbye: print "so long": return
```

Notice the "@" must precede the name both in the call and the definition and that the name is defined as a single statement. Keep in mind that redefinition errors are not checked for. The first occurrence of a label will always be used.

```
  5 rem using string variables
 10 for x = 1 to 3
 20 read g$
 30 gosub 'g$
 40 next
 50 end
 60 @xxxx: print "may not be found ";: return
 70 @yyyy: print "tokenized labels ";: return
 80 @zzzz: print "using this technique ";: return
 90 data zzzz,yyyy,xxxx
```

As it is, the above program would print (quite truthfully),

"using this technique tokenized labels may not be found".

If you were to change the data line to 90 yyyy,xxxx,zzzz then the program would print

"tokenized labels may not be found using this technique"

Notice that the '@' is used before the string variable name also.

Variables used in GOTO and GOSUB parameters must always be defined. NULL or ZERO values will not be substituted (as Basic so loves to do). Instead you will get an UNDEFINED STATEMENT error.

GOGO DANCER is only about 200 bytes of code and shouldn't be much trouble to enter as data statements. If you enter and convert the source to whatever assembler you love best, then you will be able to relocate it easily and modify it if you so desire. To activate it use:

SYS 49152

either from the immediate or within a program. A SYS49152 with the wedge enabled wont hurt anything.

GOGO DANCER was written mostly for fun and education (my own). Nonetheless, the self-modifying powers and reduction of Basic's line number dependence may prove useful in protection schemes, complex Basic intelligence simulations and in reducing the size of source programs.

Gogo Dancer 64: BASIC Loader

```
DP 100 rem save"0:gogowedge.ldr",8
EP 110 rem ** written by chris miller, kitchener, ontario
MO 120 :
BD 130 for j=49152 to 49357: read x: poke j,x
       : ch=ch+x: next
EO 140 if ch<>26694 then print "** checksum
       error **": stop
HF 150 print "sys49152: rem to enable": end
EB 160 :
AO 170 data 169,  11, 141,   8,   3, 169, 192, 141
GO 180 data   9,   3,  96,  32, 115,   0,  32,  20
BF 190 data 192,  76, 174, 167, 201, 137, 240,  43
LM 200 data 201, 141, 240,  13, 201,  64, 208,   3
ED 210 data  76, 248, 168,  32, 124,   0,  76, 237
HE 220 data 167, 169,   3,  32, 251, 163, 165, 123
EE 230 data  72, 165, 122,  72, 165,  58,  72, 165
JB 240 data  57,  72, 169, 141,  72,  32,  67, 192
PB 250 data  76, 174, 167,  32, 115,   0, 201,  64
BA 260 data 240,  12,  32, 124,   0,  32, 158, 173
JD 270 data  32, 247, 183,  76, 163, 168,  32,   6
IG 280 data 169, 136, 177, 122, 201,  36, 208,  34
EF 290 data  32, 115,   0,  32, 139, 176, 160,   0
HE 300 data 177,  71, 240,  91, 133, 255, 200, 177
CI 310 data  71, 170, 200, 177,  71, 168, 138, 208
KH 320 data   1, 136, 202, 134, 253, 132, 254,  76
AI 330 data 140, 192, 132, 255, 165, 122, 133, 253
IK 340 data 165, 123, 133, 254, 165,  43, 133,  95
MK 350 data 165,  44, 133,  96, 160,   4, 166, 255
HA 360 data 177,  95, 201,  64, 208,  25, 136, 136
PF 370 data 136, 177, 253, 200, 200, 200, 200, 209
HI 380 data  95, 208,  12, 202, 208, 240, 200, 177
OJ 390 data  95, 240,  23, 201,  58, 240,  19, 160
MJ 400 data   0, 177,  95, 170, 200, 177,  95, 133
GA 410 data  96, 134,  95, 177,  95, 208, 205,  76
CD 420 data 227, 168,  56,  76, 197, 168
```

Gogo Dancer 64: Source Code (Buddy 64 Format)

```
DI 1000 rem save"0:gogowedge.src",8
KJ 1010 rem by chris miller jan. 4, 1987
BI 1020 rem enables calculated goto(sub) ln#
GB 1030 rem e.g. n=10: goto n*n (i.e. goto 100)
EH 1040 rem enables goto(sub) labels
NO 1050 rem e.g. gosub @pig
AG 1060 rem enables goto(sub) variable name$
AF 1070 rem e.g. x$="pig": gosub @x$
AD 1080 rem e.g. 5000 @pig: (to flag routine)
KA 1090 rem ** source in buddy 64 (power
        assembler) format
AM 1100 :
KF 1110 sys999
GG 1120 ; sys700 for pal and symass
AO 1130 ;
EL 1140 *=49152
DJ 1150 .mem ; .opt oo for pal
OP 1160 ;
AF 1170 srcptr    =   $5f
LB 1180 argptr    =   253
MJ 1190 length    =   255
GC 1200 ;
IE 1210 ;** wedge for variable goto's & gosubs
KD 1220 ;
LB 1230           lda  #<wdg
AG 1240           sta  $308      ; install cmd wedge
LC 1250           lda  #>wdg
AJ 1260           sta  $309
CO 1270           rts
GH 1280 ;
KK 1290 wdg       =    *
DP 1300           jsr  $73       ;get command
ME 1310           jsr  parse
HH 1320           jmp  $a7ae
IK 1330 ;
KI 1340 parse     =    *
ND 1350           cmp  #137      ;is it goto
IB 1360           beq  goto      ;yes/then do goto
AN 1370 ;
OE 1380           cmp  #141      ;is it gosub
CE 1390           beq  gosub     ;yes/then do gosub
OO 1400 ;
GD 1410           cmp  #"@"      ;is it a label
NH 1420           bne  tobasic   ;yes/then ignore
MA 1430 ;
DJ 1440           jmp  $a8f8     ;read to eoln
AC 1450 ;
NJ 1460 ;** back to basic
CP 1470 tobasic   =    *
NN 1480           jsr  $7c       ;sets chr flags again
NN 1490           jmp  $a7ed     ;continue parsing
CF 1500 ;
GM 1510 gosub     =    *
BO 1520           lda  #3
BJ 1530           jsr  $a3fb     ;check stack space
EM 1540           lda  $7b
KI 1550           pha            ;push stuff on stack
FN 1560           lda  $7a
OE 1570           pha            ;text pointer
```

```
BO 1580            lda  $3a
OG 1590            pha            ;linenumber
NN 1600            lda  $39
GM 1610            pha
BK 1620            lda  #141
IC 1630            pha            ;gosub token
GM 1640            jsr  goto
BM 1650            jmp  $a7ae
CP 1660 ;
OF 1670 goto       =    *
PO 1680            jsr  $73       ;get next char
AC 1690            cmp  #"@"      ;see if label goto
KG 1700            beq  islabel   ;yes/then check type
EC 1710 ;
JK 1720 ;** eval expr following goto(sub)
MH 1730            jsr  $7c       ;reset flags
AJ 1740            jsr  $ad9e     ;evaluate expression
PP 1750            jsr  $b7f7     ;convert float to fixed
GF 1760 ;
CK 1770 gotoline   =    *
NA 1780            jmp  $a8a3     ;perform goto
EH 1790 ;
AO 1800 ;** label used instead of linenum
LJ 1810 islabel    =    *
IH 1820            jsr  $a906     ;index end of st'mt
MF 1830            dey            ;with .y.
BN 1840            lda  ($7a),y   ;check last char
CH 1850            cmp  #"$"      ;see if string var
BF 1860            bne  setlabel  ;no/then literal
EM 1870 ;
AO 1880 ;** else label is in a string var
CG 1890            jsr  $73
EN 1900            jsr  $b08b     ;locate var
BM 1910            ldy  #0
JB 1920            lda  (71),y    ;variable pointer
NI 1930            beq  undefnd'st ;var not defined
KA 1940 ;
NF 1950            sta  length    ;save length
II 1960            iny
OG 1970            lda  (71),y    ;set pointer to var
EC 1980            tax            ;in .x. and .y.
GK 1990            iny
DO 2000            lda  (71),y
JK 2010            tay
OB 2020            txa            ;then backup one
DF 2030            bne  for1
OG 2040 ;
NL 2050            dey
CI 2060 ;
PB 2070 for1       =    *
HN 2080            dex
HO 2090            stx  argptr    ;set zpg pointer
OO 2100            sty  argptr+1
DL 2110            jmp  hunt      ;find the line
OL 2120 ;
EO 2130 ;** otherwise label is constant
NH 2140 setlabel   =    *
HJ 2150            sty  length    ;of label argument
GI 2160            lda  $7a       ;copy txtptr
KK 2170            sta  argptr    ;to search arg pointer
EE 2180            lda  $7b
IO 2190            sta  argptr+1
OA 2200 ;
FC 2210 hunt       =    *
LE 2220            lda  43        ;read start of basic
LP 2230            sta  srcptr    ;to source pointer
EM 2240            lda  44
KC 2250            sta  srcptr+1
KE 2260 ;
EK 2270 back1      =    *
LD 2280            ldy  #4
DC 2290            ldx  length
NF 2300            lda  (srcptr),y ;first line char
BN 2310            cmp  #"@"      ;see if label line
DD 2320            bne  nextln    ;no/try next line
AJ 2330 ;
MO 2340 back2      =    *
JO 2350            dey
DP 2360            dey
NP 2370            dey
OJ 2380            lda  (argptr),y ;get search char
MM 2390 ;
MG 2400 for2       =    *
KE 2410            iny
EF 2420            iny
OF 2430            iny
IG 2440            iny
NN 2450            cmp (srcptr),y ;compare with dest
MH 2460            bne  nextln    ;not eq./next line
MB 2470 ;
HM 2480            dex            ;else continue
FB 2490            bne  back2     ;for lenght of arg
KD 2500 ;
OK 2510            iny
JC 2520            lda  (srcptr),y ;should be end
AP 2530            beq  found     ;of source label
CG 2540 ;
AE 2550            cmp  #":"
GP 2560            beq  found
AI 2570 ;
OD 2580 nextln     =    *
JG 2590            ldy  #0
AN 2600            lda  (srcptr),y
NP 2610            tax
MB 2620            iny
OO 2630            lda  (srcptr),y
AL 2640            sta  srcptr+1
EF 2650            stx  srcptr
MA 2660            lda  (srcptr),y
KN 2670            bne  back1
OO 2680 ;
BM 2690 undefnd'st =    *
AE 2700            jmp  $a8e3     ;undefined state error
MA 2710 ;
JA 2720 found      =    *
LC 2730            sec
HO 2740            jmp  $a8c5     ;set goto
ED 2750 ;
EK 2760 .end
```

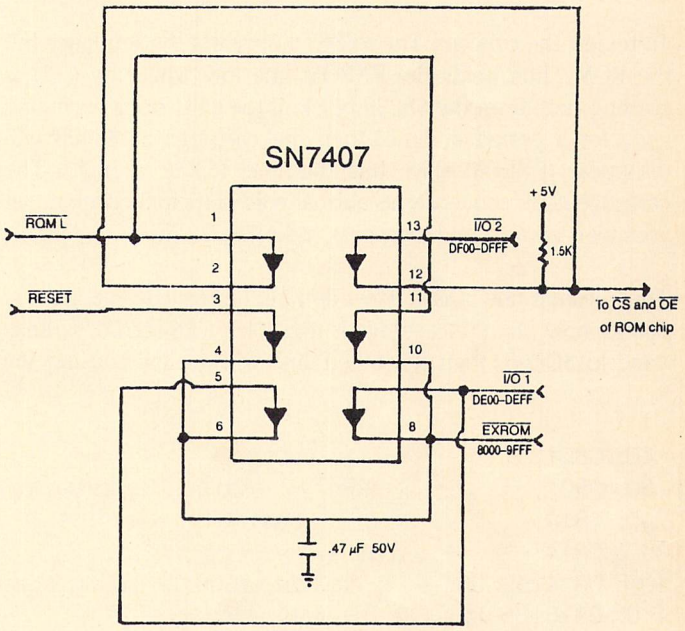
Now You See It, Now You Don't

Michael R. Mossman
Quispamsis, NB

The secret of "transparent" cartridges on the C64

Have you ever wondered why those marvellous cartridges can do things to your computer, but never show up anywhere in memory? The first time I saw Busscard II (an IEEE interface for the C64 that adds basic 4.0 disk commands) at a friends house, it made me curious. I asked to see the machine language code, but we could not find it in memory. This curiosity stayed on the back burner until I bought my Fast Load cartridge. Again, the program could not be found in memory. Overwhelmed, I proceeded to dismantle the cartridge. Inside, I expected to find a maze of modern electronics. Much to my disappointment, there sat two lowly ICs. One was the expected EPROM and the other a 7407. I traced some of the lines but it didn't make much sense. Disappointed, I closed it up and it remained in the back of my computer for over a year.

A few months back, I bought a "Promenade" EPROM programmer to burn a few of my own custom chips. Every now and then I would cast an eye at that Fast Load cartridge, wishing that I could make my cartridges invisible in memory. I revived my attack on that despicable cartridge with renewed vigor. I removed the EPROM from the board and read the program out with my Promenade. I looked at the code and figured that the program ran at $8000 but I knew that the program could not be seen at $8000. This time I sat down and traced out every line on the board and drew a diagram as I went. Low and behold, the secrets were revealed to me.



1) Pin 7 is Ground
2) Pin 14 goes to +5V supply
3) All address and data lines on the ROM or EPROM go to their equivalents on the expansion port.

Figure 1

I would like to point out that this article is not to show you how to copy the Fast Load cartridge. The cartridge is such good value for the money that building one yourself costs more than buying it outright. The code itself is of no use because it will not run by just loading and running it at $8000 – it is much more involved than that. The value lies in being able to put wedges in BASIC and set vectors that are completely transparent to other programs. All this and your program occupies no memory. The memory area at $C000 – $CFFF is fought over by so many programs. There are times when I want the DOS wedge and another program in memory at the same time. This is impossible because they conflict at $C000.

To make your own invisible program, it is necessary to understand the normal control line operation of the expansion port. These are the lines available:

EXROM – This line is normally high (1). To tell the PLA that you want the CPU to read the external rom at $8000, this line is set low.

ROML – This line is a type of decoded address line. When the CPU wants to read the external ROM at $8000, this line is pulled low or 0. ROML will never go low if the EXROM line is not low.

RESET – This line is usually high when the computer is running. Its purpose is to prevent the CPU from trying to execute ML instructions when the computer is cold started. This allows the other chips to reach their "normal" states before the CPU addresses them. RESET is low during reset time. The computer would act flaky without a RESET line. The RESET line goes low in only two normal operations:

1) When the computer is turned on.
2) When the reset button is pressed on the computer.

I/O 1, I/O 2 – These lines are intended for selecting an external I/O device. e.g. Adding an ACIA or a CIA chip. Selection is done by pulling the line low. This is done when you do a read or write to $DE00 – $DFFF. I/O 1 is the area from $DE00 – $DEFF and I/O 2 is $DF00 – $DFFF.

Now, let's look at the invisible cartridge – see Figure 1. The chip is a 7407 hex buffer and so if a low or a 1 is put in, a low or 1 comes out. When the computer is turned on, the RESET line is low. This causes the EXROM line to be low. The line is held low for a period of time, after reset, by the capacitor. When the computer reads the $8000 area it will see the EXROM line in a low state and use ROML to address the external cartridge. If it finds the autostart sequence, it then passes control over to the cartridge code. All this time, the EXROM line has been held low because ROML line is low.

To review the concept: The RESET line starts the sequence but the ROML line holds the EXROM line low while the CPU is reading code from the $8000 block. If the CPU stops executing code for a period of time, then the cartridge at $8000 will disappear (EXROM stays high because ROML is high). The cartridge code sequence is: normal cold start initialization, set your own vectors, and then pass control to BASIC.

The question now arises "How do I get the code to reappear at $8000, now that the cartridge is invisible?". If a read or write is done to $DE00, then the I/O 1 line will go low causing the EXROM line to go low. The capacitor will hold the line low for period of time. Just enough, so that when a read or write is done to $8000, the ROML will be pulled low by the CPU because EXROM is still low. In this case, I/O 1 line starts the sequence but the ROML line, again, holds it.

One of the vectors that you set in the cold start code could point to code in the cassette buffer, the $02A7 – $02FF area, or $C000 block. This code is necessary because it will make the $8000 code visible again. The drawback is that using the above areas is dangerous because other programs like to use these same spots. The answer is in using the I/O 2 line. You will notice from Fig.1 that I/O 2 is connected to the CS (selected by a low) through a buffer. When you do a read of the area from $DF00 – $DFFF you will see code. The magic thing about this code is that it is really located in the rom chip at $9F00 – $9FFF. You appear to see it at I/O 2 area because of the way the chip is selected.

Let's look at how this type of cartridge can be used in your own code. Suppose you would like to implement a wedge in basic. When the machine is turned on, the RESET line is pulled low causing the EPROM at $8000 to appear. The cartridge stays visible because of the capacitor on the EXROM line. The code at $8000 is executed because the key code exists. You make a jump to the $DFF0 area to initialize I/O devices, perform the RAM test, set up page zero kernal locations and then the I/O vectors are set. Chrget and various zero page BASIC pointers and finally the vectors at $0300 – $030B are set. It is here that you can now set the BASIC error vector at $0300 to point to your code at $DF00. When the BASIC interpreter errors out because it does not recognize a command, the error vector will point to your code at $DF00. The code at $DF00 will do a read or write to $DE00. This will cause the EXROM line to go low and the eprom to appear at $8000. You can now jump to your code in the EPROM to check the chrget routine for your command. If it is your wedge then the command is carried out, if not then you jump to the normal error handling routine.

I can see many uses for this type of programing and I think that many of you will also. Included here is a little machine language program that will make the code from the Fast-Load cartridge appear and then store it to normal ram at $8000.

```
 40: 0801                    .opt p4
 50: 0801              store =   $fb      ;address for loop storage
 60: 0814                    .bas ml
 90: 0814              ml    =   *
100: 0814 a9 00             lda  #$00     ;set up for read write loop
110: 0816 85 fb             sta  store
120: 0818 a9 80             lda  #$80
130: 081a 85 fc             sta  store+1
140: 081c a0 00             ldy  #$00
150: 081e a2 00             ldx  #$00     ;completely discharge ca-
                                          pacitor for reading eprom
160: 0820                   loop =   *
170: 0820 8e 00 de          stx  $de00
180: 0823 ca                dex
190: 0824 d0 fa             bne  loop
200: 0826                   loop1 =  *    ;loop for reading eprom
210: 0826 8c 00 de          sty  $de00
220: 0829 b1 fb             lda  (store),y ;read eprom
230: 082b 91 fb             sta  (store),y ;store to ram at same
                                          memory location
240: 082d c8                iny
250: 082e f0 03             beq  add
260: 0830 4c 26 08          jmp  loop1
270: 0833                   add  =   *    ;if low byte is zero then
                                          increase high byte by one
280: 0833 e6 fc             inc  store+1
290: 0835 a5 fc             lda  store+1
300: 0837 c9 a0             cmp  #$a0     ;if high byte is equal to
                                          $a0 then end
310: 0839 f0 03             beq  end
320: 083b 4c 26 08          jmp  loop1
330: 083e                   end  =   *
340: 083e 60                rts           ;return to basic.program
                                          can now be read with a
                                          monitor from $8000-$9fff
```

Fiddling About

Matthew Palcic
Xenia, Ohio

. . .Contrary to common belief, color graphics are easily achieved on the C128 in 80 column mode. . .

I purchased my 128 in September of 1985, immediately after they became available in my area. I was looking forward to the excellent graphics that all of the magazines talked about. I soon found out that these graphics were nothing short of impossible to figure out without any technical references. My good friend, Lowell, is a Doodle expert. After seeing the limited capability of the 128, he became very unimpressed with the 80 column graphics. I was far from convincing him that the 128 was worth the money. People soon figured out how to do hi–res in 80 columns. New drawings and utilities popped up all over.

But these programs all lacked something big: color. I still didn't know color was possible, until I fiddled around with a demo graphic screen done in 80 columns. I stopped the program and typed on the 80 column screen. A lot of trash appeared on the bitmap. I then turned on attributes, and set the attribute pointer to the same place I was typing. As I typed, I was getting different foreground colors on a black background. Lowell still felt this was nothing spectacular, because hi–res on the 64 allows a separate foreground and background. I then typed shifted and reversed characters. With that I got different background colors!! The colors worked very similar to hi–res mode of the 64. This proved that there was a systematic color scheme. I just had to figure it out. With this, Lowell challenged me to convert a Doodle picture to 80 columns in full color. And because of that challenge, I wrote this article.

Contrary to common belief, color graphics are easily achieved on the C128 in 80 column mode. The key is to shrink the size of the screen. 16 colors are allowed on the screen. Each 8 by 8 character space can support a separate foreground and background color. The colors work differently than those of the 8564 VIC–II chip used in 40 column mode. Following this article is a program that will convert a 40 column graphic screen to the 80 column screen with full color.

The first step in creating a color screen is reducing the frame size. The frame is the actual size of the screen (measured in rows and columns). For this article, a frame size of 40 columns by 25 rows is ideal. This reduces the bit map area to 8K, leaving sufficient memory for color. Because a full screen would take up 16,000 bytes of RAM, there would not be enough memory to support color with 16K of VDC RAM. Reduce the frame size by changing register one (horizontal displayed) to 40. A VIC–II graphic screen will fit perfectly in the reduced frame. The other half of the screen will then be used for color memory. For those wanting to stretch the frame to fill the screen, use double–width pixel mode. Horizontal registers will need to be adjusted when using that mode. After adjusting the screen you can then add color.

The colors are accomplished in RGB mode much differently than the composite method. In 40 column mode the colors are a set pattern; each color is assigned a bit pattern. The bit pattern is not systematic, and the values must be memorized or taken from a chart. However, RGB color can be arrived at systematically. Each of the colors in RGB (Red Green Blue) is assigned a bit. All other colors are mixtures of those primary colors. An intensity bit is also used to allow shades. The following chart explains how the bit patterns work, and how intensity affects them.

Intensity Off

Red	Green	Blue	= Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Purple
1	1	0	Brown
1	1	1	Light Grey

Intensity On

Red	Green	Blue	= Color
0	0	0	Dark Grey
0	0	1	Light Blue
0	1	0	Light Green
0	1	1	Light Cyan
1	0	0	Light Red
1	0	1	Light Purple
1	1	0	Yellow
1	1	1	White

Because of the systematic patterns in RGB, a direct transfer of composite color memory to VDC color memory would result in incorrect coloring. The following chart shows the difference between the assigned colors in composite mode and those of the RGB mode.

Composite Colors	RGB Colors
0 = Black	0 = Black
1 = White	1 = Dark Gray
2 = Red	2 = Dark Blue
3 = Cyan	3 = Light Blue
4 = Purple	4 = Dark Green
5 = Green	5 = Light Green
6 = Blue	6 = Dark Cyan
7 = Yellow	7 = Light Cyan
8 = Orange	8 = Dark Red
9 = Brown	9 = Light Red
10 = Light Red	10 = Dark Purple
11 = Dark Gray	11 = Light Purple
12 = Medium Gray	12 = Brown
13 = Light Green	13 = Yellow
14 = Light Blue	14 = Light Gray
15 = Light Gray	15 = White

Not only are the colors different in the two modes, but the foreground and background are reversed. Normal VDC attributes don't work the same as they do in bit map mode. In text mode, the bottom nibble is used to support the foreground color as it is in bit map mode. But the upper nibble is used to support a background color for a character space. These bits are normally used to support underline, flashing, reverse and alternate characters. Attributes are on by default, but to turn them on in case a program shuts them off you can set register 25, bit 6. You must also tell the VDC where in VDC RAM you want the attributes. This is done with registers 20 and 21. 20 is the high byte and 21 is the low byte of the 16 bit address.

With that background on 80 columns, you should be able to follow my conversion program. The program begins by clearing the 80 column chip with the block fill function. The destination address must be set at registers 18/19 (high byte/low byte). Then place the fill byte in register 31 (i.e. 0 to clear or 255 to fill). Next, clear bit 7 of register 24 to select block fill, and place the number of memory locations to fill in register 30. Following the initial write to register 31, there will already be one byte written. Selecting a 0 word count (reg 30) will write to 256 bytes of VDC RAM. The program then blanks the screen with a simple process involving register 35 (display enable/begin).

The next step, done with the COPY routine, involves copying color memory from $1C00–$1FFF (7168–8191) to $1300 (4864). This preserves the screen so you can compare the two modes, or make adjustments to the color table and convert again without having to reload the hi–res picture. The colors are then converted with SHIFT to an 80 column equivalent, where possible. Because 80 columns consists of 8 dark shades and 8 light shades, there wasn't an exact match for each of the composite colors. The process is to work down to a nibble (4 bits) and use that nibble as an offset to a 16 byte color lookup table. Otherwise, a 256 byte lookup table would be needed, and modifying one color would require 32 bytes to be changed to modify all occurrences of that color. The offset method requires only one byte to be changed to fix all 32. The foreground and background also need to be reversed, as they work opposite in the two modes. The following chart shows the default color conversion table. This is stored in variable COLORS and can be modified where needed.

40 Column	becomes	80 Column
00 Black		00 Black
01 White		15 White
02 Red		08 Dark Red
03 Cyan		07 Light Cyan
04 Purple		11 Light Purple
05 Green		04 Dark Green
06 Blue		02 Dark Blue
07 Yellow		13 Yellow
08 Orange		10 Dark Purple
09 Brown		12 Brown
10 Light Red		09 Light Red
11 Dark Gray		01 Dark Gray
12 Medium Gray		06 Dark Cyan
13 Light Green		05 Light Green
14 Light Blue		03 Light Blue
15 Light Gray		14 Light Gray

As you can see, no easy algorithm could make this translation, and not all colors have a perfect match, unless you would make more than one color become brown or red, etc.

Next the VIC–II screen at $2000–$3FFF (8192–16383) is sent to the VDC RAM with VICTOVDC. Because of my inexperience in machine language I made no attempt to write that routine from scratch. It is a modified version of the routine found in the 128 Programmer's Reference Guide (Bantam Books). For more details on that type of process, see Paul Durrant's program and article in the September issue. (Games from the Inside Out)

After the hires screen is transferred, the color can be put in. The HITME routine simply sets the data pointer (reg's 18/19) to $2000 VDC (attributes) and copies the translated colors from $1300. The screen is then turned back on with UNBLANK and. . .voila!

The process is very quick, especially if fast mode is used. As the machine language is one program, you can easily use it with other Basic programs, etc. I hope that you will not merely use the routine as it is, but will experiment with the possibilities proven. The potential is even greater if you replace the 16K chips with 64K chips. My friend, Lowell, is now working on getting a 128 and will also install the 64K chips as I have done. He needed no further proof that the 128 in 80 columns can do INCREDIBLE color graphics. I'll cover 64K in my next article. Until then, I strongly urge you to play around with all the registers to see what you can do. After all, look where it got me.

Basic demo For Color 80

```
KM 100 rem save"0:viewer.bas",8
NG 110 rem ** color 80  viewer **
JM 120 rem **  matthew palcic  **
GP 130 :
CB 140 graphic 1,1: graphic 5,1: sys dec("c027")
       : print"go to 40 cols": graphic 0,1
JJ 150 if peek(dec("d00"))<>198 then gosub 270
       : rem position code
LE 160 print "file to load (or $ for dd directory)"
MA 170 input ">";f$
HK 180 if f$ = "$" then scnclr: directory "dd*": print
       : goto 160
JH 190 if f$ = "*" then goto 220
PO 200 scnclr: graphic 1
LL 210 bload(f$),p7168
JF 220 fast: sys dec("c00"): slow
LJ 230 getkey a$: graphic 1,1
NJ 240 goto 160
IH 260 :
HH 270 for j = 3072 to 3522: read x: poke j,x
       : ch = ch + x: next
FH 280 if ch<>48663 then print "** checksum
       error **": stop
OD 290 return
AK 300 :
PE 310 data  32,  22,  12,  32, 174,  13,  32, 146
BL 320 data  12,  32,  61,  12,  32, 183,  12,  32
HF 330 data 122,  13,  32, 187,  13,  96, 162,  18
AP 340 data 169,   0,  32,  96,  13, 232,  32,  96
MA 350 data  13, 169,   0,  32,  94,  13, 162,  24
JI 360 data  32, 110,  13,  41, 127,  32,  96,  13
IM 370 data 160,  64, 162,  30, 169,   0,  32,  96
BN 380 data  13, 136, 208, 250,  96, 169,   0, 133
BN 390 data 250, 169,  19, 133, 251,  32,  88,  12
BK 400 data 230, 251,  32,  88,  12, 230, 251,  32
JN 410 data  88,  12, 230, 251,  32,  88,  12,  96
CK 420 data 160,   0, 177, 250, 133, 252,  74,  74
OB 430 data  74,  74, 170, 189, 130,  12, 133, 253
HN 440 data 165, 252,  10,  10,  10,  10,  74,  74
NI 450 data  74,  74, 170, 189, 130,  12,  10,  10
MH 460 data  10,  10, 101, 253, 145, 250, 200, 208
HH 470 data 217,  96,   0,  15,   8,   7,  10,   4
FM 480 data   2,  13,  11,  12,   9,   1,   6,   5
CN 490 data   3,  14, 169,   0, 133, 250, 133, 252
EG 500 data 169,  28, 133, 251, 169,  19, 133, 253
CA 510 data 162,   4,  32, 173,  12, 230, 251, 230
JK 520 data 253, 202, 208, 246,  96, 160,   0, 177
EC 530 data 250, 145, 252, 200, 208, 249,  96, 162
HL 540 data  25,  32, 110,  13,   9, 128,  32,  96
GG 550 data  13, 162,  20, 169,  32,  32,  96,  13
MI 560 data 162,  21, 169,   0,  32,  96,  13, 162
JO 570 data  12, 169,   0,  32,  96,  13, 162,  13
IO 580 data 169,   0,  32,  96,  13, 162,   1, 169
DE 590 data  40,  32,  96,  13, 169,  32, 133, 251
HD 600 data 169,   0, 133, 250, 162,  18,  32,  96
EJ 610 data  13, 232,  32,  96,  13, 133, 177, 169
ID 620 data  25, 133, 155, 169,   7, 133, 156, 169
CL 630 data  39, 133, 254, 162,   0, 161, 250,  32
LM 640 data  94,  13,  32,  61,  13, 198, 254, 208
KB 650 data 242, 165, 177, 208,  21, 162,   0, 161
LN 660 data 250,  32,  94,  13,  32,  73,  13, 198
LB 670 data 156, 208, 220, 169,   1, 133, 177,  76
EJ 680 data 255,  12, 169,   0, 133, 177, 162,   0
FA 690 data 161, 250,  32,  94,  13,  32,  87,  13
KB 700 data 198, 155, 208, 191,  96,  24, 165, 250
EL 710 data 105,   8, 133, 250, 144,   2, 230, 251
LP 720 data  96,  56, 165, 251, 233,   1, 133, 251
EC 730 data 165, 250, 233,  55, 133, 250,  96, 230
HM 740 data 250, 208,   2, 230, 251,  96, 162,  31
DO 750 data 142,   0, 214,  44,   0, 214,  16, 251
KO 760 data 141,   1, 214,  96, 162,  31, 142,   0
BC 770 data 214,  44,   0, 214,  16, 251, 173,   1
FE 780 data 214,  96, 162,  18, 169,  32,  32,  96
EM 790 data  13, 162,  19, 169,   0,  32,  96,  13
NH 800 data 169,   0, 133, 250, 169,  19, 133, 251
BB 810 data  32, 163,  13, 230, 251,  32, 163,  13
JJ 820 data 230, 251,  32, 163,  13, 230, 251,  32
PD 830 data 163,  13,  96, 160,   0, 177, 250,  32
KF 840 data  94,  13, 200, 208, 248,  96, 162,  35
NB 850 data  32, 110,  13, 133, 255, 169,   0,  32
AH 860 data  96,  13,  96, 162,  35, 165, 255,  32
LG 870 data  96,  13,  96
```

Source Code For Color 80

```
JD 1000 rem save"0:color 80.src",8
PA 1010 rem ** source start-up in power assembler
        format (aka buddy-128 system)
EC 1020 sys4000
KA 1030 * = $0c00
EL 1040 .mem
AJ 1050 ;
DJ 1060 ; color 80 color hi-res
AF 1070 ; composite to rgb converter
MK 1080 ; matthew palcic - 16k v2.1
IL 1090 ;
PL 1100          jsr   clear    ;main subroutine
KG 1110          jsr   blank    ;table
KG 1120          jsr   copy
DB 1130          jsr   shift
BD 1140          jsr   victovdc
CK 1150          jsr   hitme
CA 1160          jsr   unblank
OH 1170          rts
CB 1180 ;
KE 1190 clear    =     *        ;clear vdc w/block fill
GE 1200          ldx   #18      ;set data pointer to 0
ME 1210          lda   #0       ;for start of hires
IK 1220          jsr   writer
KK 1230          inx
ML 1240          jsr   writer
FE 1250          lda   #0       ;set fill byte to 0
MC 1260          jsr   write
PB 1270          ldx   #24      ;clear bit 7 of
EI 1280          jsr   reeder   ;register 24 to
FJ 1290          and   #127     ;select block fill
IP 1300          jsr   writer
LC 1310          ldy   #64      ;clear 64 pages (64*256 =
                                 16384 bytes to clear)
PK 1320          ldx   #30      ;reg 30 is word count
BE 1330          lda   #0       ;0 words = 256 bytes
```

```
CL 1340 ;
AL 1350 clear1  =   *
ED 1360         jsr writer
FB 1370         dey
IE 1380         bne clear1
EO 1390 ;
EG 1400         rts
IP 1410 ;
KE 1420 shift   =   *       ;translate color info
BI 1430         lda #0
JL 1440         sta $fa
NF 1450         lda #$13    ;color start $1300
EI 1460         sta $fb     ;convert $1300
GO 1470         jsr conv    ;
LH 1480         inc $fb     ;convert $1400
OA 1490         jsr conv
AJ 1500         inc $fb     ;convert $1500
CC 1510         jsr conv
FK 1520         inc $fb     ;convert $1600
GD 1530         jsr conv
AP 1540         rts
EI 1550 ;
DL 1560 conv    =   *
NG 1570         ldy #0
CK 1580 ;
BH 1590 nibbles =   *       ;convert foreground
KF 1600         lda ($fa),y ;get color
JG 1610         sta $fc
NN 1620         lsr         ;shift foreground down
PH 1630         lsr         ;to bottom nibble
BE 1640         lsr
LE 1650         lsr
HE 1660         tax
PN 1670         lda colors,x ;get new value
OF 1680         sta $fd     ;store foreground
KD 1690         lda $fc     ;retrieve original color byte
BG 1700         asl         ;shift left to clear top nibble
JF 1710         asl
DG 1720         asl
NG 1730         asl
PK 1740         lsr         ;move back to bottom nibble
PK 1750         lsr
JL 1760         lsr
DM 1770         lsr
PL 1780         tax
HF 1790         lda colors,x ;get new value
NM 1800         asl         ;move background to top nibble
NL 1810         asl
HM 1820         asl
BN 1830         asl
DL 1840         adc $fd     ;combine foreground and
                            background
CL 1850         sta ($fa),y
EC 1860         iny
NE 1870         bne nibbles
EE 1880         rts
IN 1890 ;
CA 1900 colors  =   *       ;40 col color
KE 1910         .byte 0     ;black
OA 1920         .byte 15    ;white
KC 1930         .byte 8     ;red
JH 1940         .byte 7     ;cyan
OC 1950         .byte 10    ;purple
AL 1960         .byte 4     ;green
FI 1970         .byte 2     ;blue
KF 1980         .byte 13    ;yellow
IO 1990         .byte 11    ;orange
DH 2000         .byte 12    ;brown
PH 2010         .byte 9     ;light red
BM 2020         .byte 1     ;dark gray
GK 2030         .byte 6     ;medium gray
HI 2040         .byte 5     ;light green
DB 2050         .byte 3     ;light blue
NJ 2060         .byte 14    ;light gray
MI 2070 ;
EJ 2080 copy    =   *
FB 2090         lda #0
OK 2100         sta $fa     ;copy color ram
EO 2110         sta $fc     ;from normal vic-ii
GJ 2120         lda #$1c    ;screen ($1c00)
OG 2130         sta $fb
AH 2140         lda #$13    ;copy to $1300
II 2150         sta $fd
ON 2160         ldx #4      ;copy 4 pages
AP 2170 ;
IL 2180 copy1   =   *
JI 2190         jsr copyit
GJ 2200         inc $fb
GK 2210         inc $fd
DG 2220         dex
OJ 2230         bne copy1
GD 2240 ;
GL 2250         rts
KE 2260 ;
MO 2270 copyit  =   *
DD 2280         ldy #0
IG 2290 ;
EJ 2300 copybyte =  *
AE 2310         lda ($fa),y
AJ 2320         sta ($fc),y
KP 2330         iny
PN 2340         bne copybyte
EK 2350 ;
EC 2360         rts
IL 2370 ;
CG 2380 victovdc =  *       ;translate vic-ii hires to vdc hires
IB 2390         ldx #25     ;set register 25
KJ 2400         jsr reeder  ;bit map mode
OO 2410         ora #128    ;bit 7
IF 2420         jsr writer
CI 2430         ldx #20     ;set attributes at $2000 (vdc ram)
BL 2440         lda #$20
GH 2450         jsr writer
KH 2460         ldx #21
BJ 2470         lda #0
EJ 2480         jsr writer
FD 2490         ldx #1      ;set vdc screen width (reg 1)
JO 2500         lda #40     ;to 40 columns
CL 2510         jsr writer
MF 2520         lda #$20    ;start of vic-ii hires $2000
OP 2530         sta $fb
HN 2540         lda #0
PA 2550         sta $fa
IP 2560         ldx #18     ;data pointer (vdc) to $0000
KG 2570         jsr writer  ;for start of vdc bit map
AP 2580         inx
CA 2590         jsr writer
PE 2600         sta $b1     ;column counter
MH 2610         lda #$19
OD 2620         sta $9b
ML 2630 ;
HJ 2640 again1  =   *
CC 2650         lda #7      ;8 bytes per character (0-7)
JG 2660         sta $9c
EO 2670 ;
OH 2680 again   =   *
MK 2690         lda #$27    ;40 columns (0-39)
BL 2700         sta $fe
```

```
MA 2710 ;
PG 2720 tranhi   =   *
BP 2730          ldx #0
JL 2740          lda ($fa,x)
OP 2750          jsr write
CD 2760          jsr add
EL 2770          dec $fe
LC 2780          bne tranhi
MF 2790 ;
DJ 2800          lda $b1
CE 2810          bne half
KH 2820 ;
FF 2830          ldx #0
NB 2840          lda ($fa,x)
CG 2850          jsr write
LM 2860          jsr increase
IP 2870          dec $9c
GM 2880          bne again
AM 2890 ;
BE 2900          lda #1
PD 2910          sta $b1
HC 2920          jmp again
IO 2930 ;
NK 2940 half     =   *
BH 2950          lda #0
BH 2960          sta $b1
BO 2970          ldx #0
JK 2980          lda ($fa,x)
OO 2990          jsr write
GF 3000          jsr upthere
BI 3010          dec $9b
EL 3020          bne again1
ME 3030 ;
MM 3040          rts
AG 3050 ;
MF 3060 add      =   *
EH 3070          clc
DO 3080          lda $fa
PP 3090          adc #8
FD 3100          sta $fa
IN 3110          bcc add1
GK 3120 ;
ID 3130          inc $fb
KL 3140 ;
FE 3150 add1     =   *
EE 3160          rts
IN 3170 ;
MI 3180 increase =   *
HP 3190          sec
OF 3200          lda $fb
HI 3210          sbc #1
AL 3220          sta $fb
JH 3230          lda $fa
AA 3240          sbc #$37
LM 3250          sta $fa
IK 3260          rts
MD 3270 ;
AK 3280 upthere  =   *
FN 3290          inc $fa
OO 3300          bne up1
EG 3310 ;
GP 3320          inc $fb
IH 3330 ;
EH 3340 up1      =   *
CA 3350          rts
GJ 3360 ;
MM 3370 write    =   *       ;write to vdc ram
EB 3380          ldx #31
EL 3390 ;
IC 3400 writer   =   *       ;write to vdc register
CJ 3410          stx $d600
CN 3420 ;
DB 3430 write1   =   *
ME 3440          bit $d600   ;wait for status bit to go high
LO 3450          bpl write1
KP 3460 ;
JO 3470          sta $d601   ;write value
EI 3480          rts
IB 3490 ;
BJ 3500 reed     =   *       ;read vdc ram
GJ 3510          ldx #31
GD 3520 ;
LM 3530 reeder   =   *       ;read vdc register
EB 3540          stx $d600
EF 3550 ;
LN 3560 reed1    =   *
OM 3570          bit $d600   ;wait for status bit to go high
MI 3580          bpl reed1
MH 3590 ;
FL 3600          lda $d601   ;read value
GA 3610          rts
KJ 3620 ;
AL 3630 hitme    =   *       ;transfer color from ram to vdc
LO 3640          ldx #18     ;data pointer to $2000 (attributes)
LG 3650          lda #$20
AD 3660          jsr writer
KE 3670          ldx #19
LE 3680          lda #0
OE 3690          jsr writer
PF 3700          lda #0
HJ 3710          sta $fa
DD 3720          lda #$13    ;colors are at $1300
CJ 3730          sta $fb     ;transfer $1300
JF 3740          jsr ahead
LI 3750          inc $fb     ;transfer $1400
NG 3760          jsr ahead
CK 3770          inc $fb     ;transfer $1500
BI 3780          jsr ahead
JL 3790          inc $fb     ;transfer $1600
FJ 3800          jsr ahead
OM 3810          rts
CG 3820 ;
DP 3830 ahead    =   *
LE 3840          ldy #0
AI 3850 ;
CF 3860 ahead1   =   *
GG 3870          lda ($fa),y ;read ram
CC 3880          jsr write   ;write to vdc
CB 3890          iny
IA 3900          bne ahead1
ML 3910 ;
MD 3920          rts
AN 3930 ;
OH 3940 blank    =   *       ;blank 80 cols
KF 3950          ldx #35
BP 3960          jsr reeder
GC 3970          sta $ff     ;store current screen params
HH 3980          lda #0
IA 3990          jsr writer  ;blank it
MI 4000          rts
AC 4010 ;
DO 4020 unblank  =   *       ;restore screen
KK 4030          ldx #35
GG 4040          lda $ff     ;retrieve screen params
AL 4050          jsr writer  ;restore param
IM 4060          rts
MF 4070 ;
MM 4080 .end
```

Twin–80 Screen For the Commodore 128

D.J. Morriss
Toronto, Ontario

Use the extra VDC memory for another 80–column screen

The 8563 Video Display Controller (VDC) chip that controls the 80–column screen on the C–128 has its own RAM memory completely outside the C–128 memory space. The VDC uses 8 K bytes to store the complete 512–member character set, 2 K bytes for screen memory, and another 2 K bytes for attribute memory (one byte of each for each screen location in the 80 column by 25 line screen). Thus, a complete screen and character set requires 12 K bytes. Since 12 K byte chips are somewhat scarce, the VDC in fact has 16 K bytes of RAM available. Various uses have been proposed for this extra memory: a tiny RAM disk, for example. Coincidentally, the unused 4 K bytes are exactly what is needed for a completely separate screen memory and attribute memory. This program sets up such a screen, and allows you to toggle between the two screens, in either immediate or program mode.

What Must Be Done

In order to support such a double screen, both the VDC and the screen editor must know where, in VDC RAM, screen memory and attribute memory starts for each of the two screens. The VDC needs to know where to look for information when it is drawing the screen, and the screen editor needs to know where to put information in response to PRINT commands. The VDC looks to its own internal registers $0C and $0D to contain the address, in VDC RAM, of the start of screen memory, while registers $14 and $15 contain the address of the start of attribute memory. Both these register pairs are in high–byte, low–byte order; just the reverse of normal 8502 order. The default 80–column screen uses the first 2 K bytes of VDC RAM for screen memory, and the second 2 K byte block for attributes. I have chosen to leave these untouched, and set up the second screen memory in the unused region from 4 K to 6 K, and the second screen attributes from 6 K to 8 K; above 8 K, character RAM starts. So to inform the VDC about the new memory allocation, all that is necessary is to toggle VDC register $0C between the values 0 and $10, and toggle VDC register $14 between the values $08 and $18.

The screen editor must also know where to put things. It relies on C–128 RAM location $0A2E, which contains the page of the start of screen memory in the VDC RAM, and location $0A2F, which contains the page of the start of attribute memory in VDC RAM. Since only the page is stored, the screen editor insists that both screen memory and attribute memory start on page boundaries, although the VDC does not make the same demand. So to inform the screen editor about the new memory allocation, it is only necessary to toggle $0A2E between the values 0 and $10, and toggle $0A2F between the values $08 and $18.

It is also necessary to reset values in other VDC registers and in C–128 RAM. These locations contain information about cursor location, windows that are enabled, locations of tab stops, quote mode, number of inserts, type and size of cursor etc. Specifically, locations from $E0 to $F9, $0354 to $0361, and $0A2B in C–128 RAM all contain information that defines the precise nature of a screen. These are the locations that the screen–editor routine SWAPPER switches when you transfer between 40 and 80–column screens. Similarly, in the VDC, registers $0A, $0B, $0E, $0F, $18, $1A, and $1D contain information that also defines the precise nature of a screen. When you toggle between 80– column screens, the values in all these locations must be stored, while stored values are placed in all these locations.

What Does It

The switch between screens is triggered by a custom ESCape sequence. There are twenty–seven (twenty–eight?) default ESCape sequences programmed into the C–128 ROM; they add a lot of power and convenience to the screen editor. Commodore also designed–in flexibility; at the point in the ESCape sequence handling routine where the next key after the ESC has been detected and stored in the accumulator, the ROM routine jumps through an indirect address in RAM at $0338 – $0339, before implementing the default ESC sequences. Resetting this vector to point to a new routine opens up the possibility of two hundred and fifty–six different ESCape sequences!

In this application, the sequence ESC – UP ARROW is used to toggle between the two screens. As is the case with the normal ESCape sequences, this means pressing and releasing the ESC key, then pressing the UP ARROW key. In program mode, PRINTing CHR$(27) and an UP ARROW (or CHR$(94)) will accomplish the same effect. The UP ARROW is not a cursor key; it's the exponentiation symbol, between the RESTORE and the ASTERISK keys.

If you prefer some other sequence, look at the sixth number in the fourth DATA statement in the BASIC loader program,

RELOCATING/TWIN. It's 94, the decimal CHR$ value for the UP-ARROW. Replace this one byte with the CHR$ code for the character of your choice, and the change is made. If you choose one of the default ESCape sequences, this custom application will take precedence, as long as you are in 80-column mode, since this special handling routine comes before the default routine.

What Do You Get

The two 80 - column screens available with this program are completely equal and independent. Each screen has its own tab settings, colors, cursor type, and windows. Either screen can be set to reverse or normal mode, or cleared, independently of the other. When you toggle between the two screens, the cursor returns to the spot (and window) where it was when you left the screen. If the cursor for that screen was a non-flashing underline when you left, it will be the same when you come back. If you have used some of the VDC registers to change the size of the block cursor, that new cursor will remain in effect for the particular screen in which it was set up. If you are in quote mode, typing a program line when you toggle out, you will be in quote mode when you toggle back. In fact, it's easy to lose track of which screen you're looking at. Location $0A2E is zero for the normal screen, and $80 if the new screen is active. But there is no real reason to worry about this, since THE TWO SCREENS ARE COMPLETELY EQUIVALENT!!

In one important way, this double screen arrangement is superior to the windows available on the C-128. If you list a long program line, one that runs to two or more physical lines, the screen editor makes a note of this in what is called a line-link map, found in C-128 RAM from $035E to $0361. If you create, and then collapse a window, this line-link map is destroyed. If you were to press RETURN on one of these long lines, only the one physical line would be recognized. When you toggle between the Twin-80 screens, however, the line-link map is stored and replaced with the appropriate one from storage.

Who Needs It

If you want to be able to display twice as much in the way of results from a program, you need it. If you want a program to PRINT constant progress reports, while results are left secure and unscrolled on the other screen, you need it. If you would like to look at two parts of a long program listing at the same time, you need it. If you're using the TRACE utility, and would like to list various lines of the program, without destroying the TRACE values, you need it. If you have a HELP screen, full of information that you keep thumbing through manuals for, you certainly need it. Or if you just hate the idea of 4,096 bytes of memory going to waste and never being used, you need it.

The Programs

RELOCATING/TWIN

Free RAM in Bank 15, where this routine is located, is getting cluttered up with various custom routines, which always seem to compete for the same space. To alleviate this problem somewhat, this BASIC loader for TWIN-80 is a relocating one. The program asks you where in memory to put itself, then puts itself there and activates itself - if you just press RETURN in response to this prompt, the program will start at location 6144, which should be safe. The bytes that must be adjusted for each new starting point are flagged by minus values in the DATA statements. The program does all the adjusting; just be careful not to miss any minus signs. There are many possible locations in Bank 15 RAM (see Table 1); surely they won't all be occupied by some other important utility. If you are really tight for space, the first 32 bytes of the routine are needed only for installation, not for the actual routine.

Although the entire routine runs to 224 bytes, there are only 206 values in the DATA statements. Each of the 18 negative values generates two bytes to be POKEd into memory.

Remember, if you choose to use a different ESCape sequence to switch screens, both the VERIFIZER report code for the DATA line, and the total checksum, CK, will be off.

TWIN-80.ASM

This program, my first with a symbolic assembler, was written using the "Power Assembler" (formerly "Buddy") 128 Macro-Assembler from Spinnaker. As an alternative to doing the same job using the built-in Monitor assembler, Buddy leaves me ecstatic, verging on euphoric. If you have a different assembler, you should have no trouble converting the power assembler source code to work with it; the most likely thing that you'll have to change are the long symbol names with the embedded apostrophes. So, for example, the label "save'bank" could be changed to "SAVBNK" to work with another assembler.

The set-up routine simply resets the ESCVEC vector, temporarily sets Bank 15, momentarily activates both screens to clear them of garbage, and restores the bank setting before exiting.

The new ESCape handler checks for the special ESCape sequence, and that you are using the 80-column screen, before switching all the necessary locations.

Communication with the VDC is aided by the two ROM routines, READREG at $CDDA and WRITEREG at $CDCC. Both routines move information between the accumulator and the VDC register whose number is found in the X register. The routines take care of the "handshaking" between the C-128 and its independent-minded VDC.

What Next

Clearly, we haven't heard the last of multiple screens for the VDC. For example, if you are willing to live with one colour for the entire screen, no under-lining, no flashing, and only one character set, you can disable the attributes. Then the two, 2 K byte blocks of attribute memory would be freed up to provide room for two more independent screens; a total of four. Since disabling the attributes eliminates one of the character sets, this means half of the 8 K of character RAM is available; two more screens, for a total of six! Or 150 lines of 80-column text, just a key-stroke or two away!

Table 1

AVAILABLE RAM IN BANK 15

Location (Dec)	Purpose
1024 – 2047	40–column screen
2816 – 3071	Cassette buffer
3072 – 3327	RS–232 Input buffer
3328 – 3583	RS–232 Output buffer
3584 – 4095	Sprite storage area
4864 – 7167	Applications Program area
7168 – 16383	High Resolution Screen (after GRAPHIC 1)

Twin–80: Relocating BASIC Loader

```
OC 1000 rem *** relocating loader for "twin–80" ***
LF 1010 rem *** by d. j. morriss, toronto, ontario ***
OP 1020 fast : ck=0
IG 1030 input"starting location (decimal)[7 spaces]6144
       [6 left]"; ad : if ad>16165 then 1030
NJ 1040 for k=ad to ad+223
NM 1050 read x: ck=ck+x : if x=>0 then poke k, x
       : goto 1080
AJ 1060 hb=int((ad+abs(x))/256) : lb=ad+abs(x)–256*hb
GG 1070 poke k, lb : k=k+1 : poke k, hb
MD 1080 next
KM 1090 if ck <> 15424 then print"*** error in data
       statements ***": stop
EI 1100 hb=int((ad+32)/256) : poke ad+6, hb
MK 1110 lb=ad+32–256*hb : poke ad+1, lb
PG 1120 sys ad
AI 1130 print:print" twin–80 screen activated!"
CK 1140 print:print" press escape, then up–arrow, to toggle
       between screens!"
OH 1150 print:print" bsave from "ad" to "ad+223" (decimal)
       to save obj program!"
II 1160 end
HL 1170 data 169, 32, 141, 56, 3, 169, 19, 141
GK 1180 data 57, 3, 32,–121, 169, 147, 32, 210
KK 1190 data 255, 32,–140, 169, 147, 32, 210, 255
NE 1200 data 32,–140, 76,–133, 201, 94, 240, 3
PA 1210 data 76, 193, 201, 165, 215, 41, 128, 240
MN 1220 data 247, 32,–121, 32,–140, 173, 43, 10
LN 1230 data 174,–167, 142, 43, 10, 141,–167, 162
FK 1240 data 25, 181, 224, 188,–169, 157,–169, 148
AI 1250 data 224, 202, 16, 243, 162, 13, 189, 84
ED 1260 data 3, 188,–195, 157,–195, 152, 157, 84
CO 1270 data 3, 202, 16, 240, 160, 6, 190,–209
AD 1280 data 32, 218, 205, 72, 185,–216, 32, 204
KN 1290 data 205, 104, 153,–216, 136, 16, 236, 76
FC 1300 data –133, 173, 0, 255, 141,–168, 169, 0
OP 1310 data 141, 0, 255, 96, 173,–168, 141, 0
CG 1320 data 255, 96, 173, 46, 10, 73, 16, 141
DD 1330 data 46, 10, 162, 12, 32, 204, 205, 173
AG 1340 data 47, 10, 73, 16, 141, 47, 10, 162
EE 1350 data 20, 32, 204, 205, 96, 96, 0, 0
AI 1360 data 16, 0, 18, 24, 0, 0, 79, 0
CH 1370 data 0, 3, 0, 0, 24, 79, 94, 27
OE 1380 data 7, 7, 0, 0, 0, 0, 0, 0
PH 1390 data 0, 128, 128, 128, 128, 128, 128, 128
CC 1400 data 128, 128, 128, 0, 0, 0, 0, 10
HI 1410 data 11, 14, 15, 26, 24, 29, 160, 231
NM 1420 data 16, 0, 240, 32, 231, 255
GJ 1430 end
GK 1440 scratch"relocating/twin":dsave"relocating/twin"
```

Twin–80: Source Code in Buddy 128 Format. Load address: 7169

```
GP 1000 scratch"twin/80":dsave"twin/80":end
IG 1010 ;
CH 1020 ;
CI 1030 ;      twin–80.asm
GI 1040 ;
NF 1050 ;   submitted by: d. j. morriss
LO 1060 ;            769 coxwell avenue
AL 1070 ;            toronto ontario
PK 1080 ;            m4c 3c6
LI 1090 ;            (416) 466 2791 (home)
PC 1100 ;            (416) 967 1212 (work)
NF 1110 ;               ext 3276
II 1120 sys4000
KO 1130 ;***** variable table ******
IO 1140 :
CG 1150 mode      =   $d7        ;bit 7 = 1 for 80 column
DE 1160 pnt       =   $e0        ;start of zero page screen parameters
LD 1170 escvec    =   $0338      ;location of escape routine wedge
JE 1180 tabmap    =   $0354      ;start of active screen tabs and line links
PF 1190 curmod    =   $0a2b      ;shadow for vdc reg # $0a
EP 1200 vm3       =   $0a2e      ;ram shadow for hi byte of vdc
KA 1210 :                        ;start of screen
OA 1220 vm4       =   $0a2f      ;ram shadow for hi byte of vdc
MD 1230 :                        ;start of attributes
JD 1240 escape    =   $c9c1      ;normal escape handling routine
IG 1250 writereg  =   $cdcc      ;writes to vdc register
HA 1260 readreg   =   $cdda      ;reads from vdc register
HJ 1270 mmucr     =   $ff00      ;find bank setting here
FN 1280 jbsout    =   $ffd2      ;kernal print routine
AH 1290 table'1'length = 26      ;number of values in table 1
EH 1300 table'2'length = 14      ;number of values in table 2
NB 1310 table'3'length =  7      ;number of values in tables 3 and 4
MJ 1320 :
DD 1330 ;***** end of variables *****
EB 1340 .org $1300               ;assembles into empty ram
KO 1350 .mem
BE 1360 ;*** set–up routine ***
OM 1370 :
NP 1380 :         lda  #<new'handler  ;reset escape vector
NH 1390 :         sta  escvec         ;to point to new
JM 1400 :         lda  #>new'handler  ;routine
BJ 1410 :         sta  escvec+1
AA 1420 :
DA 1430 :         jsr  save'bank   ;save bank and set bank 15
EB 1440 :
NF 1450 :         lda  #$93        ;clear low screen
PD 1460 :         jsr  jbsout
CD 1470 :
CK 1480 :         jsr  vdc'toggle  ;configure for high screen
GE 1490 :
NK 1500 :         lda  #$93        ;clear high screen
BH 1510 :         jsr  jbsout
EG 1520 :
CP 1530 :         jsr  vdc'toggle  ;configure for low screen and
EE 1540 :         jmp  old'bank    ;restore bank setting
DA 1550 :                          ;using rts to get to basic
MI 1560 :
HB 1570 ;** end of set–up routine **
AK 1580 :
GM 1590 ;** new ecape handling routine *
EL 1600 :
LI 1610 new'handler =  *
LB 1620 :         cmp #$5e         ;is it up arrow
AE 1630 :         beq do'it        ;yes, so go to special routine
EJ 1640 norm'out  jmp  escape      ;no, so go normal route
GO 1650 :
```

```
JN 1660 do'it     lda  mode          ;is 80-column being used
KH 1670 :         and  #%10000000
CO 1680 :         beq  norm'out      ;no, so go normal route
OA 1690 :
BB 1700 :         jsr  save'bank     ;save bank and go bank 15
CC 1710 :
CJ 1720 :         jsr  vdc'toggle    ;switch memories
GD 1730 :
CH 1740 :         lda  curmod        ;switch curmod and table0
PG 1750 :         ldx  table0
JH 1760 :         stx  curmod
BM 1770 :         sta  table0
IG 1780 :
CN 1790 :         ldx  #table'1'length-1  ;interchange values in
GA 1800 loop1     lda  pnt,x              ;table 1 with page 0
FO 1810 :         ldy  table1,x           ;from $e0 to $f9
OP 1820 :         sta  table1,x
CP 1830 :         sty  pnt,x
MN 1840 :         dex
LG 1850 :         bpl  loop1
IL 1860 :
BG 1870 :         ldx  #table'2'length-1  ;interchange the tab maps
PL 1880 loop2     lda  tabmap,x           ;and line link maps
DK 1890 :         ldy  table2,x           ;with values in table 2
BF 1900 :         sta  table2,x
LI 1910 :         tya
IL 1920 :         sta  tabmap,x
GD 1930 :         dex
IM 1940 :         bpl  loop2
CB 1950 :
DI 1960 :         ldy  #table'3'length-1
GF 1970 loop3     ldx  table3,y      ;interchange the values
LH 1980 :         jsr  readreg       ;in table 4 with the
IO 1990 :         pha                ;values in selected vdc
LO 2000 :         lda  table4,y      ;registers
KM 2010 :         jsr  writereg      ;registers are listed in
EK 2020 :         pla                ;table 3
KN 2030 :         sta  table4,y
FK 2040 :         dey
JD 2050 :         bpl  loop3
AI 2060 :
AP 2070 :         jmp  old'bank      ;get old bank, and use its
BL 2080 :                            ;rts to return to basic
ED 2090 ;*** end of new handler routine ****
IK 2100 :
FE 2110 ;** subroutines ****
JP 2120 ;** routine to stash bank & set bank to 15 **
ND 2130 save'bank lda  mmucr         ;save bank in
GM 2140 :         sta  table0+1      ;table 0, then set
OA 2150 :         lda  #$00          ;to bank 15
KC 2160 :         sta  mmucr
HI 2170 :         rts
IP 2180 :
PF 2190 ;**** routine to fetch and reset old bank ****
AK 2200 old'bank  lda  table0+1      ;get old bank
DM 2210 :         sta  mmucr         ;and store in mmu
JL 2220 :         rts
KC 2230 :
IM 2240 ;* routine to toggle between screens in vdc memory **
FM 2250 vdc'toggle lda  vm3          ;toggle value of vm3
BE 2260 :         eor  #$10          ;between $00 and $10
LD 2270 :         sta  vm3
EC 2280 :         ldx  #$0c          ;store new value in
JD 2290 :         jsr  writereg      ;vdc register # $0c
AH 2300 :
BM 2310 :         lda  vm4           ;toggle value of vm4
NK 2320 :         eor  #$10          ;between $08 and $18
LH 2330 :         sta  vm4
FF 2340 :         ldx  #$14          ;store new value in
KG 2350 :         jsr  writereg      ;vdc register # $14
FE 2360 :         rts
GN 2370 ;**** end of subroutines ****
AM 2380 :
DK 2390 table0    =    *
DN 2400 .byte 96,0
KL 2410 table1    =    *
KG 2420 .byte 0,16,0,18,24,0,0,79,0,0,3,0,0,24,79,94,27,7,7,0,0,0,0,0,0,0
BN 2430 table2    =    *
CF 2440 .byte 128,128,128,128,128,128,128,128,128,128,0,0,0,0
IO 2450 table3    =    *
AI 2460 .byte 10,11,14,15,26,24,29
PP 2470 table4    =    *
GD 2480 .byte 160,231,16,0,240,32,231
GJ 2490 .end
```

Memory Lane

Chris Miller
Kitchener, ON.

Psssst, . . .wanna know a really sneaky place to stash stuff in the 128?

By now the MMU (Memory Management Unit) at $FF00 in the C128 is fairly well understood. But there is more, a lot more involved in really being able to stroll down its memory lane (without getting lost or tired or mugged).

Under-the-Rug

Pssst! Hey Mac (Maxine) wanna know a really sneaky place to stash stuff in the 128? Okay, just step around the corner into this here alley.

Enter the following code through the monitor – put it at $140.

```
0140 sei          ;no interruptions please
0141 lda #7e
0143 sta ff00     ;ram 1 with i/o visible
0146 lda #f1
0148 sta d508     ;bit 0 at $d508 is the key to this trick
0151 lda #20
0153 sta d507     ;swap pages $00 and $20
0156 ldx #2       ;$00 and $01 don't swap
0158 sta 2000,x   ;store blanks at "original" zero page
                   loop
0161 inx
0162 bne 0158
0164 lda #0       ;restore the registers
0166 sta d508
0169 sta d507
0172 sta ff00
0175 brk          ;return to monitor
```

G 140, then try to find the 254 $20's. Betcha can't. Okay, now use the same setup but LDA 2000,X then STA 8000,X in the looping part. Now after G 140 you will see that 8002-80ff (in bank 1) is all spaces. So those spaces must have been somewhere, otherwise how could you get at them again?

Way back when I was still an un-enlightened goof I thought I had a back door to 8563 RAM. Now that I am an enlightened goof I believe that these bytes are sitting at 02-ff in RAM 1 (not shared RAM 0).

A Zero Page of Your Very Own

One of the neatest features of the 128 is its relocatable, or more precisely, swappable zero page and stack areas. In the above example page $00 and page $20 are swapped. A write to $ff would show up at $20ff in shared RAM 0 (always RAM 0) but a write to $20ff would appear at $ff in unshared RAM 1.

Normal practice (mine anyway) involves writing only to the low bytes ($D507 and $D509) of the page pointer registers. Then, new and old system pages will always be in shared RAM 0 which even overides the $ff00 memory configuration setting and gives rise to a slick way of accessing any page of RAM 0 from RAM 1.

Here an Interrupt, There an Interrupt, Everywhere an Interrupt

Here is a handy little routine to make assembly language programming in out-of-the-way banks a bit more homey on the 128. Brian Hilce, author of Pro-Line's Cocompiler, put me on to it so that I would stop griping to him about what a bother it was trying to write code to run in Bank 1. The following IRQ/BRK wedge allows interrupts to be serviced no matter which bank of memory one's code happens to be executing in.

First point the BRK vector at the normal IRQ service routine. Then just point the IRQ vector at your own routine which must be in common RAM. Normally this will be below $400. All you do when an interrupt hits your routine is put a $00 at $ff00 (Bank 15) and execute a BRK instruction followed by a NOP. Execution resumes following the NOP. The original $FF00 and register values will be on the stack-restore them. An RTI gets you back completely intact to wherever you came from. Sixty times a second; works like a charm. Looks like this:

```
        sei
        lda 788       ;first point brk vec
        sta 790       ;at irq service code
        lda 789
        sta 791
        lda #<irqsrv  ;then point irq vec
        sta 788       ;at your switch
        lda #>irqsrv
        sta 789
        cli
        rts

irqsrv  =*            ;interrupt wedge
        lda #0
        sta $ff00     ;set bank 15 config
        brk           ;service irq thru brk
        nop
        pla           ;old config on stack
        sta $ff00     ;restore it
        pla           ;registers too
        tay
```

```
        pla
        tax
        pla
        rti             ;return from interrupt
```

The same general idea could be used in the 64 to take advantage of interrupt servicing from the hidden RAM below Kernal ROM.

If your program was using its own zero page you might want to push the interrupted $D507 setting on the stack and stick a $00 there prior to the BRK so that the IRQ routine would not mess with your pointers, then, right after the BRK:NOP, pull $D507 off the stack and reset it. Fancy that – over 250 bytes of wonderful, beautiful, absolutely free and uninterrupted zero page for your program.

Keeping in Touch With the Kernal

Even though the 128 has a bunch of new Kernal routines for calling out–of–bank code, I tend to shy away from them. These tend to jump all over the place, slowing things down and generally confusing the daylights out of me. Besides, it's usually the Kernal I'm calling in the first place. I like to write my own JSRFAR routine to handle Kernal calls from other banks. Usually there is room low on the stack ($140–$1C0) or in the system input buffer ($200–$29f) for this sort of thing. The following lets you call the Kernal from Bank 1. One byte of memory is needed to save the .Y register.

```
;.Y is saved in HOLDY
; and loaded with the low byte of the Kernal routine
; prior to calling FARKERNAL

FARKERNAL =*
        sty  kernrtn  ;self mod jsr
        ldy  #0
        sty  $ff00    ;bank 15
        sty  $d507    ;your own Z page Mr. Kernal sir.
        ldy  holdy    ;in case it's needed
        jsr  $ffff    ;call routine
kernrtn =*-2
        php           ;save status
        pha           ;and .A
        lda  #myzpg;;my zero page back
        sta  $d507
        sta  $ff02;   ;latch to bank 1
        pla           ;remember .A
        plp           ;and status
        rts
```

Here is how FARKERNAL would be used to PRINT from Bank 1:

```
print   =*
        sty  holdy    ;save .y
        ldy  #$d2     ;<$ffd2
        jmp farkernal
```

If you were converting a program from the 64, this print routine would replace the PRINT = $FFD2 assignment. The rest of your program should never know the difference.

Notice again that your very own, personal, private, exclusive zero page will still remain that way. Neither interrupts nor Kernal calls need be allowed to chew on it.

Share and Share Alike

The system keeps a $04 at $d506 that provides an essential 1K of common RAM at the bottom of memory. So normally, no matter what bank you do business with, common RAM from $00 to $3ff will be visible. Almost every big ML program is going to have to move a little code into this area in order to access all of memory. Unhappily, the operating system hogs a good deal of it. But despair not, for this area can be expanded and even moved about at will. Overseeing this is the *RAM configuration* register at $D506. Only the low nybble is used in the 128. The two low order bits determine the amount of RAM to share. Paired values of 0,1,2 or 3 result in 1K, 4K, 8K or 16K respectively of shared RAM. Bits 2 and 3 determine whether low, high, or both low and high RAM is held common.

Common RAM is always RAM 0. The $D506 setting takes priority over the $FF00 configuration but not over the page pointers. If you just can't seem to fit all your relay code below $400, or if you want to be able to access the 40 column screen directly from Bank 1, then keep $D506 in mind.

There is another situation in which expanding common RAM is virtually essential. ROM for the Z80 CPU resides from $00–$fff. At first glance it would seem impossible to switch Banks in Z80 mode without whipping memory right out from under you. No common RAM to work from. . . unless you were to first store, say, a %1011 at $D506, locking the top 16K ($C000–$FFFF) into RAM 0.

That Secret Place Again

Writing a $00 to $D506 disables all common RAM (except $ff00) including the first K of memory. It is virtually impossible to change banks in this mode. The 128 monitor is unable to access $00–$3ff RAM 1 for this very reason. But now this perfectly free well–hidden K of RAM is yours for the taking by simply storing a $00 at $D506 anywhere from within RAM 1.

Beyond the MMU

ML memory management of the 128 goes well beyond massaging $FF00 directly or via its $FF01–$FF04 write latches to values at $D501–$d504. However, using these in conjunction with the page pointers ($D507–$D508 and $D509–$D50A) and the RAM CONFIGURATION register ($D506) will pave the way to tapping the real potential of the versatile 128.

Why Me?

I have just finished (for the 100th time) new 64/128 assemblers (my Buddy–Systems) for Pro–Line software so I would like to think I know a great deal about the ins and outs of programming the 128. I would also like a Jaguar XKE, a satellite dish and a cottage by the lake.

Event Maker for the Amiga

by Chris Zamara, Technical Editor

– Simulate mouse and keyboard inputs with CLI commands

One of the first things you'll notice about any computer is how you have to ask it to do the things you want done. In computerese, that translates to the "User Interface". User interfaces come in two flavours: command–driven, and interactive. The Amiga, being the do–everything machine that it is, lets you have it both ways.

From the CLI (Command–Line Interface), the command–happy folks can merrily punch in obscure incantations to set the machine into action on any mission they wish to assign it. Programs designed to be used from the CLI are controlled by the arguments typed in after the program name, making a single command. The arguments can be names of files or other data for the program to work on, or certain options and settings that the program looks at to find out what you expect from it. The built–in CLI commands like Copy, Format, Echo, Search, etc. can all be used in this manner, and many other programs get their arguments from the command line in this way as well. All the Unix–style utilities that programmers are so fond of are operated via command–line arguments.

The Workbench, of course, is an example of interactive program operation. Although Workbench uses a Window/Icon/Mouse/Pointer kind of user–interface, it isn't just for WIMPs, and even Real Programmers can use it to good effect (although they usually do so only when no one's looking). Interactive interfaces like Workbench are typically easier to use than command–driven ones, and can often be used without relying on reference manuals or your memory. An example of an operation that can be performed via a command OR interactively (using Workbench) is copying a file. You can either type "copy file1 to file2" from CLI, or drag an icon (if there is one for that file) from one disk or drawer to another on the Workbench screen.

Naturally, each approach lends itself better to some applications than others. A graphics editor, for example, should be interactive, while a file converter of some kind would probably be more useful as a command (programs that operate either way are a boon, but that's not always practical). The command–driven program has a major advantage over the interactive one, though: it can be operated from an "Execute", or "script", file. A script file can automate some process that the user would otherwise have to carry out by typing in commands one at a time, possibly waiting for operations to complete in between. The script file just contains a list of commands as they would be typed from CLI, and is executed with the CLI Execute command. An interactive program will be of little use in such a process, since it will come up and wait for the user to enter something, which makes the operation no longer automatic.

A real example? You just used "PageSetter" by Gold Disk, Inc. to create a lovely report, complete with the obligatory pie–charts and bar graphs that everyone uses to show off their computer. Now, you want to print several copies of this report, perhaps advancing the paper between copies. You want this to happen without your attention, since it may take a long time. No problem – use the "PagePrint" program provided on the PageSetter disk to print your PageSetter file. PagePrint is excellent in that it can be run either from Workbench, or from the CLI by giving it the name of the PageSetter file you want to print.

Fine – make a script file to perform the PagePrint commands, with Echo commands thrown in to advance the page after each copy is printed. Execute the script file and away you go. Except, oh look – PagePrint puts up a window and asks you to press RETURN to print the document, or ESCAPE to abort. So much for your automatic script file – you'll have to stick around while it runs and press RETURN every time PagePrint is run to produce the next copy. This is not to say anything bad about PagePrint – it's just designed as an interactive program, rather than being completely command driven. You will probably find many programs that you would like to be able to operate from a script file (even from a Startup–Sequence), but they require keyboard or mouse input from the user before they will do their job and finish up so that the script file can continue. What you need in such instances is a way to make the computer in effect "press its own buttons".

Which, not surprisingly, is why we are here today. We present to you "Eventmaker", a command–driven program that lets you simulate interactive input. It works like this: you tell Eventmaker what input events you wish to take place, such as mouse movements, key presses and releases, etc., and those events happen within the Amiga as if the user actually performed them. This can get spooky, since you can have a program do things by itself as if someone was operating it, somewhat like a player piano.

How To Use Eventmaker

When you run Eventmaker from the CLI, you give it a list of events you want it to simulate. The simplest of these is a keypress. For example, if a program wants you to press "c" to continue, you can get Eventmaker to do it for you: first run the program (using "run" from CLI), then get Eventmaker to enter a "c" like this:

```
Eventmaker c
```

You could enter several keys like this:

```
Eventmaker a b c d
```

Function keys are signified by "F" followed by the function key number (F1, F2, etc.).

Only unshifted characters may be specified in this way. If SHIFT, CTRL, ALT, or other keys are to be held while these keys are

pressed, they must be entered as separate events beforehand. Special keys are entered as follows (they must be in uppercase):

shift/alt etc.:
LSHIFT RSHIFT LALT RALT LAMIGA RAMIGA CTRL CAPS

mouse buttons : function keys :
LMB RMB F1 – F10

other keys:

HELP	RETURN	UP	DOWN	LEFT	RIGHT
ESC	TAB	SPACE	ENTER	DEL	BACKSPC

Key releases are signified by using the "^" character ("caret", shift–6) instead of the minus sign. For example:

Eventmaker SHIFT a ^SHIFT ^a

(press shift, press 'a', release SHIFT, release 'a')

Key releases do not normally have to be specified, unless this is important to the program receiving the input.

The left and right mouse buttons are handled just like other keys (with the events LMB and RMB), but mouse movements are handled differently. Mouse movements are preceded by "m:" (m colon), and followed by a relative mouse movement in pixels. The mouse movement is two positive or negative numbers separated by a comma that specify how many pixels to move the mouse pointer in the X and Y directions, in that order. The pixels are always in terms of a hi–res interlaced screen, that is, 640 across by 400 down. The other mouse event command that is often useful before a mouse movement is "m:h", which moves the mouse pointer "home", to the top left hand corner of the screen. By giving an "m:h" event followed by a mouse movement of the form "m:<x>,<y>", you can put the mouse pointer anywhere you want on the screen with Eventmaker. Some examples:

```
Eventmaker m:100,20          ;move pointer 100 right, 20 down
Eventmaker m:–50,0           ;move pointer 50 left
Eventmaker m:h m:320,200     ;put pointer at centre of screen
```

Sometimes a delay is needed between events; for example, when running a program by double–clicking and then trying to operate the program itself. You need to wait a few seconds while the program comes up before you can give it input. A delay can be specified between input events with the syntax "w:<n>", where n is the number of seconds to delay (within a second of accuracy). For example, here's the command you'd use to double–click on the icon beneath the mouse pointer, wait three seconds, then send a few characters to the program just run:

Eventmaker LMB ^LMB LMB ^LMB w:3 h i , SPACE p r o g r a m .

Two other events are "diskinserted" and "diskremoved", which signify the obvious. These events are specified with "d:i" and "d:r" respectively.

To simulate keys on the numeric keypad, use the prefix "n:", for example, n:0, n:–, etc. This does not apply to the ENTER key, which is signified by the word ENTER alone.

Don't worry about memorizing all of the above instructions; a summary is printed out when you run Eventmaker without giving it any arguments, i.e. just type "Eventmaker" from the CLI.

How It Works

There's nothing tricky about "Eventmaker"; it just uses a provision the system has for adding events to the input stream. All programs receive events from the input stream unless they are working at a very low level, like reading the keyboard matrix directly from the "keyboard.device", which is unlikely.

Input events are added using the "input.device", one of the many software entities in the Amiga known as "devices". A command called "IND_WRITEVENT" is given to the input device, along with an "InputEvent" structure that specifies what events are to take place. These events can be the press or release of any key on the keyboard along with SHIFT, CTRL, either ALT, or either "Amiga" command key; any mouse movement (specified in relative X and Y units); or the press or release of either mouse button.

The program opens the "input.device", then goes through the list of arguments supplied by the user, one at a time. Each argument is used to determine how to fill in an "InputEvent" structure, which is linked in to the I/O request structure sent to the input device. The WRITEVENT command is then put into the I/O request structure and given to the Input Device via the DoIO() function. The input device then enters this event into the input stream for us, and the program currently receiving input gets the event. Finally, the input device is closed and the program exits. It's all done in about 260 lines of C code, and the hardest part is parsing the user's arguments and turning them into InputEvents in the format that the input device wants them.

Things to do with Eventmaker

Being able to automate your script files with interactive programs will probably be a great help to you, but you can go further than that. Why not make a script file to:

- Create a picture with a graphics program
- Open the Workbench "Demos" drawer and run all the programs
- Run your word processor and set the options as you like them
- Start a terminal program and get it to automatically log you onto an online service

So, even if you didn't have a crying need for this program before, you can still have fun with it.

P.S.

You can use Eventmaker to find the "Easter eggs" in Workbench as described in the bits column without playing a "solitaire game of Twister". A word of warning, though: this will give you the most deeply buried message, the one you should only see if you're not offended by rude words. Just make your CLI window shorter and move it to the bottom of the Workbench screen to display the screen title bar, then use Eventmaker like this:

```
Eventmaker m:h m:600,22 LMB ^LMB m:0,–10
LSHIFT LALT RSHIFT RALT F1 d:r LMB d:i
```

```
/* Eventmaker
 * Chris Zamara, 1987
 *
 * Puts the specified character into the input stream as if it were
 * typed from the keyboard. Useful if you are running a program from a
 * script that wants you to press return or something to start.
 *
 * See the "docs" array below for complete specs.
*/

#include <exec/devices.h>
#include <devices/inputevent.h>
#include <devices/input.h>

char *docs[] = {
  "Eventmaker accepts any number of arguments,",
  "each specifying an input event. The syntax for an event is:\n",
  "<key>          - key down",
  "^<key>         - key up",
  "n:<key>        - numeric keypad key",
  "m:<x>,<y>      - relative mouse movement",
  "m:h            - mouse home (to 0,0)",
  "d:i            - disk inserted",
  "d:r            - disk removed",
  "w:<n>          - wait n seconds\n",
  "keys are entered as single unshifted characters, except:\n",
"shift/alt etc.     : LSHIFT RSHIFT LALT RALT LAMIGA RAMIGA CTRL CAPS",
"mouse buttons      : LMB RMB",
"function keys      : F1 - F10",
"other keys         : HELP  RETURN  UP      DOWN  LEFT  RIGHT",
"                     ESC   TAB     SPACE   ENTER DEL   BACKSPC",
};
struct IOStdReq *idReq = NULL, *CreateStdIO(); /* for I/O requests */
struct Port *idReqPort = NULL, *CreatePort();
long idError = 1, OpenDevice();

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

 if (argc < 2)
 { /* print instructions */
   for (i = 0; i < sizeof(docs) / sizeof(char *); i++)
     puts(docs[i]);
   exit(0);
 }
 if (OpenInputDevice() == 0)
   for (i = 1; i < argc; i++)
     MakeEvent(argv[i]);
 else
   puts("Something went wrong. Sorry, can't help you.");
 CloseInputDevice();
}

OpenInputDevice ()
/* create port and I/O request block, open input device */
{
 if ((idReqPort = CreatePort("idReqPort", 0L)) == NULL
   || (idReq = CreateStdIO(idReqPort)) == NULL
   || (idError = OpenDevice("input.device", 0L, idReq, 0L)) != NULL
   )
   return 1;
 return 0;
}

CloseInputDevice ()
{
 if (idError == 0) CloseDevice(idReq);
 if (idReqPort)    DeletePort(idReqPort);
 if (idReq)        DeleteStdIO(idReq);
}

MakeEvent (string)
/* Fill in InputEvent structure and give WRITEVENT command to input device */
char *string;
{
struct InputEvent MyInputEvent;

 idReq->io_Command = IND_WRITEEVENT;
 idReq->io_Flags   = 0;
 idReq->io_Length  = sizeof(struct InputEvent);
 idReq->io_Data    = (APTR)&MyInputEvent;

 MyInputEvent.ie_NextEvent          = NULL;
 MyInputEvent.ie_TimeStamp.tv_secs  = 0;
 MyInputEvent.ie_TimeStamp.tv_micro = 0;
 MyInputEvent.ie_X = 0;
 MyInputEvent.ie_Y = 0;

 if (SetupEvent(string, &MyInputEvent))
   DoIO(idReq);
}

parseXY (string, x, y)
/* get x and y value from user's input and put into supplied args */
char *string;
WORD *x, *y;
{
int yarg;

 *x = atoi(string);
 if ((yarg = findchar(string, ',')) != -1)
   *y = atoi(string + yarg + 1);
}

SetupEvent (eventstring, event)
/* fill in supplied InputEvent structure based on supplied event string */
char *eventstring;
struct InputEvent *event;
{
UWORD class, code, qual, mqual, nqual;
UWORD key_up_indicator = 0;
static UWORD qualifier = 0;

 code = qual = mqual = nqual = 0;
 class = IECLASS_RAWKEY;

 if (eventstring[0] == '^')     /* key up indicator - skip over */
 {
   key_up_indicator = IECODE_UP_PREFIX;
   eventstring++;
 }
 if (eventstring[1] != ':')     /* normal key event */
   code = findkeycode(eventstring, &qual, &class);
 else                           /* disk, keypad or wait */
   switch (eventstring[0])
   {
     case 'n': /* numeric keypad */
       nqual  = IEQUALIFIER_NUMERICPAD;
       code   = findkeycode(eventstring, &qual, &class);
       break;
     case 'm': /* mouse movement */
       mqual  = IEQUALIFIER_RELATIVEMOUSE;
       code   = IECODE_NOBUTTON;
       class  = IECLASS_RAWMOUSE;
       if (eventstring[2] == 'h') /* mouse home (top left) */
         event->ie_X = event->ie_Y = -1024;
       else
         parseXY(eventstring + 2, &event->ie_X, &event->ie_Y);
       break;
     case 'd': /* disk event  */
       if (eventstring[2] == 'i')
         class = IECLASS_DISKINSERTED;
       else if (eventstring[2] == 'r')
         class = IECLASS_DISKREMOVED;
       break;
```

```
        case 'w': /* wait n seconds */
            Delay((ULONG)(atoi(eventstring + 2) * 50));
            return 0;
        }
    if (class == IECLASS_RAWKEY && code > 0xFF)
    {
        printf("Bad event specification: %s\n", eventstring);
        return 0;
    }
    if (class == IECLASS_RAWKEY)
        if (key_up_indicator)
            qualifier &= ~qual;
        else
            qualifier |= qual;
    event->ie_Class     = class;
    event->ie_Code      = code | key_up_indicator;
    event->ie_Qualifier = qualifier | mqual | nqual;

    return 1;
}

findkeycode (key, qualifier, class)
/* given a single key event, find its code and qualifier */
char *key;
UWORD *qualifier, *class;
{
UWORD code;
int table_index;
char *kstring =
  "`1234567890-=\@@@qwertyuiop[]@@@@asdfghjkl;'@@@@@@zxcvbnm,./";
static char *keytable1[] = {
      "HELP",  "RETURN",  "UP",      "DOWN",  "LEFT",  "RIGHT",
      "ESC",   "TAB",     "SPACE",   "ENTER", "DEL",   "BACKSPC",
      NULL };
static UWORD code_table1[] = {
      0x5f,  0x44, 0x4c, 0x4d, 0x4f,  0x4e,
      0x45, 0x42, 0x40, 0x43, 0x46, 0x41  };
static char *keytable2[] = {
      "LSHIFT",  "RSHIFT",  "LALT",  "RALT",
      "LAMIGA", "RAMIGA", "CTRL", "CAPS",
      NULL };
static UWORD code_table2[] = {
      0x60, 0x61, 0x64, 0x65, 0x66, 0x67, 0x63, 0x62 };
static UWORD qualifier_table[] = {
      IEQUALIFIER_LSHIFT,     IEQUALIFIER_RSHIFT,
      IEQUALIFIER_LALT,       IEQUALIFIER_RALT,
      IEQUALIFIER_LCOMMAND,   IEQUALIFIER_RCOMMAND,
      IEQUALIFIER_CONTROL,    IEQUALIFIER_CAPSLOCK };
static UWORD keypad_codes[] = {
      0x4a, 0x3c, 0x00, 0x0f,  0x1d, 0x1e, 0x1f,
      0x2d, 0x2e, 0x2f,  0x3d, 0x3e, 0x3f   };

    *qualifier = 0;
    *class = IECLASS_RAWKEY;

    if (strlen(key) == 1)
        code = findchar(kstring, *key);   /* single-char key   */
    else if ((table_index = findstring(keytable1, key)) >= 0)
        code = code_table1[table_index];  /* other key         */
    else if ((table_index = findstring(keytable2, key)) >= 0)
    {                           /* qualifier-type key */
        code = code_table2[table_index];
        *qualifier = qualifier_table[table_index];
    }
    else if (*key == 'F' && strlen(key) <= 3)
        code = 0x4f + atoi(key + 1);      /* function key      */
    else if (strcmp("LMB", key) == 0)
    {                           /* left mouse button */
        code = IECODE_LBUTTON;
        *qualifier = IEQUALIFIER_LEFTBUTTON;
        *class = IECLASS_RAWMOUSE;
    }
    else if (strcmp("RMB", key) == 0)
    {                           /* right mouse button */
        code = IECODE_RBUTTON;
        *qualifier = IEQUALIFIER_RBUTTON;
        *class = IECLASS_RAWMOUSE;
    }
    else if (key[0] == 'n' && key[1] == ':' /* numeric keypad key  */
          && key[2] >= '-' && key[2] <= '9')
        code = keypad_codes[key[2] - '-'];
    else
        code = -1;                  /* invalid event specifier */

    return code;
}

findchar (string, chr)
/* find a character in a string and return its position */
char *string;
char chr;
{
int pos;

    for (pos = 0; *string != chr && *string; pos++)
        string++;

    return (*string ? pos : -1);
}

findstring (array, string)
/* find a string in an array and return its position */
char **array, *string;
{
int pos = 0;

    while (array[pos] && strcmp(array[pos], string))
        pos++;

    return (array[pos] != NULL ? pos : -1);
}
```

A New ECHO: An AmigaDOS Command Replacement

Neal Bridges, Welland, Ontario

. . .I've discovered the joys of programming in 68000 assembly language. . .

I moved from a C64 environment to an Amiga only a couple of months ago, and it's only in the last couple of weeks that I've discovered the joys of programming in 68000 assembly language. As my first project, rather than creating something totally new, I decided to improve on something already in existence. What follows is a replacement for the AmigaDos command "ECHO". This new ECHO command is shorter, faster, and better, while still retaining all the features of the old ECHO command. It works under both Release 1.1 and Release 1.2 of the Kickstart/ Workbench disks.

Three file listings follow this article.

Listing 1 ("Echo.asm") is the Commodore Macro Assembler source listing of the new command.

Listing 2 ("Echo.gen") is an AmigaBasic program generator that will generate an executable command called "ECHO" on your AmigaBasic default directory. This is for those of you lacking the Assembler. Each DATA line ends with a checksum, so the READ loop at the top can tell you what line an error is found in.

Listing 3 ("DateChange") is an example of a command file using the new ECHO.

How To Use the New ECHO

To make effective use of the new ECHO, I recommend that you directly replace the old ECHO command (in the c: directory of your CLI/Workbench disk) with the new one. . . all your existing command files will function perfectly, albeit slightly faster.

Usage: ECHO [-n] <string>

The new ECHO has three new features. In the examples that follow, 'ECHO' represents the new ECHO command file.

1) If the characters '-n' precede the <string> argument, a newline character will not be printed after the <string>. For example, within a command sequence,

```
ECHO -n Bro
ECHO ken
```

would display:

```
Broken
```

2) Quotes around the <string> are normally unnecessary and may be left out. Thus,

```
ECHO "Hello World?"
```

will display the same thing as

```
ECHO Hello World?
```

The only useful purpose they can serve is to indicate leading blanks, as in

```
ECHO "    This is 3 spaces over."
```

or to visually delimit trailing blanks (for easier editing), as in

```
ECHO -n "The current date is: "
```

Also,

```
ECHO ""
```

prints a blank line.

Apart from this, quotes are ignored by the new ECHO command.

3) Within the <string>, any character preceded by a '^' is converted into its equivalent CONTROL character. This feature makes it possible to send ANSI codes to the console and printer. For example,

```
ECHO -n ^l
```

will generate and display an ASCII 12 (Formfeed), effectively clearing the screen, and

```
ECHO >prt: ^[[1mHello There^[[0m
```

will print "Hello There" in bold type on the printer (^[results in an ESCape character).

The new ECHO handles the '*' character in exactly the same manner as the old ECHO.

About The Program

The file "ECHO.asm", when assembled and linked, comes to 292 bytes. This is 268 bytes smaller than the existing command. I happen to like short code. My first machine (way back in '82) was a 1K ZX81, so I learned to squeeze my programs right from the start. It's healthy mental exercise.

I have done my best to document the assembler listing fully. In terms of function, it first opens the DOS library. Then, if the argument was '?', it prints the template (': ') and gets an argument string from the current input device. It then checks the beginning of the argument string for a '–n'. If found, a flag is set indicating to the output section that no End–of–Line character is to be transmitted after the output string. Then the main loop copies the rest of the argument string to the output buffer, handling quotes and the special characters ('*' and '^'). It then calls the Write routine in the DOS library to send the result out to the current output device, with or without a linefeed, as indicated by the End–of–Line flag. Afterwards, the DOS library is closed, and control is returned to the CLI.

The routine is position independent. I haven't tested it, but this should mean that it loads and executes slightly faster than a similar relocatable routine would because (a) registers are faster than memory, and (b) the relocating loader in ROM doesn't have to relocate anything when it reads the command from the disk. I ran a rough test which showed that the new ECHO command is approximately 60% faster when running it from disk than the old ECHO. Being position independent makes it smaller, as well. Machine instructions that operate solely on internal registers take up less bytes than those that reference immediate values and memory addresses.

One interesting note: Rather than create a buffer to hold the output string, I re–used the argument line buffer. Since the argument line isn't referenced elsewhere after this routine is through with it, there's no problem. This only works because the length of the output string that this routine produces is always less than or equal to the length of the original argument string. If it were possible for the output string to be longer than the argument string, disaster would result because the output string would write over input characters that hadn't been scanned yet.

Another note: Although this new ECHO command does all the things the old one does, the inverse is not true. That is, if you make use of the improved features in your command files, you must have the new ECHO command in your c: directory when you EXECUTE those files. If you come up with a useful command file using the new ECHO, and you want to distribute it, don't forget to distribute the new ECHO command with it, or you'll get complaints.

Listing 1: Source Code

```
;---
;- "Echo.asm"
; A new ECHO command by Neal Bridges, 02/87.
;
; My address:
;
; Neal Bridges,
; 4 Crescent Drive,
; Welland, Ontario,
; Canada, L3B 2W5.
; Phone: (416)-734-7789.
;
;- This is a CLI command. It's meant to replace the existing
; Echo command. Although it has more features, it is upwardly compatible
; with the existing command. It's also quite a bit shorter & faster.
; To use it effectively, I recommend directly substituting
; it for the existing "Echo" command in the c: directory of your
; CLI/Workbench disk.
;
;- Usage: Echo [-n] <string>
; Notes: If you include the '-n', no linefeed will be printed at the
;      end of the line. The 'n' must be lowercase.
;
;- Special features within <string>: A '^' followed by a character
; will be converted into a CTRL character. For example, ^l becomes
; ASCII 12, (Formfeed), ^g becomes ASCII 7 (Bell), ^j becomes
; ASCII 10 (Linefeed), ^[ becomes ASCII 27 (Esc), etc.
;   Example: Echo -n ^l
;   clears the screen.
;   Example: Echo ^[[2mHello There!
;   switches to text colour number two (normally black) & displays
;   'Hello There!' in that colour.
;
;- Quotes (") around the argument <string> are unnecessary
;  unless you wish to have leading spaces in your string,
;  for example: Echo "      Hello World!"
;
;  As described in the AmigaDos Manual:
;  If for some reason you wish to print a quote character, you may do so
;  by preceeding it with a '*'.
;   Example: Echo *"Hello World.*"
;   displays: "Hello World"
;
;- If you wish to print a '*', use two of them.
;   Example: Echo Input**Output 2 3 4
;   displays: Input*Output 2 3 4
;
;- 'Echo' by itself does nothing.
;- 'Echo "" ' prints a blank line.
;
;- Permission to copy but not to sell.
;--

;-- The following register equates
;  save memory & increase the speed of the program.


;-- Register Equate:   -- Used for:
bufptr      equr a2     ;Output buffer ptr.
argpstr     equr a3     ;Storage place for ptr
                        ; to start of argument line.
argptr      equr a5     ;Ptr to start of
                        ; argument line.
char        equr d0     ;Char being read in
                        ; from argument line.
BitFlags    equr d1     ;Bit 0 is used to signal type of line termination.
                        ;Bit 1 is used for the quote
                        ;toggle flag, a thing incorporated to kludge
                        ;the strange parsing of the existing Echo command.
Specflag    equr d2     ;Flag for use with special
                        ; symbols '*' & '^'.
count       equr d3     ;Len of output line.
arglen      equr d4     ;Len of argument line.
Dosbase     equr d5     ;Dos library pointer.
output      equr d7     ;CLI outhandle.

;-- Symbolic equates:
AbsExecBase equ  $4     ;Loc of ptr to the
                        ; Exec library.
LF          equ  10     ;Line Feed value.
CTRLmask    equ  %00011111  ;Mask to make CTRL chars.
```

```
;
xlib    MACRO   ;Macro to save my feeble fingers.
        xref  _LVO\1
        ENDM
;
call    MACRO   ;Another macro to save typing.
        jsr   _LVO\1(a6)
        ENDM
;
        xlib  Close         ;These are Library routines referenced
        xlib  OpenLibrary ;externally by this program.
        xlib  CloseLibrary
        xlib  Output
        xlib  Write
        xlib  Input
        xlib  Read
;
;-- Initial setup:
_main   move.l  a0,argptr       ; Remember ptr to argument line.
        move.l  a0,argpstr      ; Make copy of it for reference.
        move.l  d0,arglen       ; Remember len of argument line.
        lea     dosname(pc),a1  ;-Specify name 'dos.library'.
        clr     d0              ; Specify any version (0 means
                                ; any version is OK).
        move.l  AbsExecBase,a6  ; Then using Exec library,
        call    OpenLibrary     ; Open Dos library.
        move.l  d0,dosbase      ; Remember dosbase ptr.
        beq     close           ; Check for error (d0=0 means
                                ; "Oh dear, it didn't open").
;
;-- Get CLI outhandle:
getout  move.l  dosbase,a6      ; Using Dos library,
        call    Output          ; get CLI outhandle,
        move.l  d0,output       ; & then remember it.
;
;-- Check the existence of an argument line:
checklen cmpi.b #1,arglen       ; If no argument chars,
        ble     close           ; exit program.
;
;-- Check for argument='?'+LF:
checkqm cmpi.w  #'?'*256+LF,(argptr) ; If the argument is '?' only,
        bne.s   strtscan        ; show template & read argument
                                ; line from console input device.
                                ; (The equation results in a word
                                ; equalling a '?' char followed by a LF).
;
;-- Display template:
getline lea     template(pc),a2 ;-Specify ptr to template string in
        move.l  a2,d2           ; d2 (via a2);
        move.l  output,d1       ; Specify outhandle in d1;
        move.l  #endtemp-template,d3 ;Specify number of chars in d3;
        move.l  dosbase,a6      ; then using Dos library,
        call    Write           ; show the template.
;
;-- Get argument line from input device:
        call    Input           ; Still using Dos library, get the
                                ; current inhandle.
        move.l  d0,d1           ;-Specify inhandle in d1;
        move.l  argptr,d2       ; Specify argument buffer ptr in d2;
        move.l  #255,d3         ; Specify max len in d3;
        call    Read            ; still using Dos library, Read line.
        move.l  d0,arglen       ; Set arglen to the actual len of the
        bra.s   checklen        ; line read & go back up to check
                                ; line len again.
;
;-- Start the argument line scanning process:
strtscan clr.b  BitFlags        ; Clear end-of-line flag and quote
                                ; toggle flag.
        move.l  argptr,bufptr   ; Set bufptr to start of
                                ; output buffer. (Since the argument line
                                ; isn't needed twice, I'm using it
                                ; for the output buffer as
                                ; well, & therefore setting
                                ; bufptr to argptr achieves my purpose.)
;
;-- Check for '-n' at start of argument line:
        cmpi.w  #'-n',(argptr)+ ; Are first 2 chars '-n'?
        bne.s   fixptr          ; If YES,
        moveq   #1,BitFlags     ; set Bit 0 of BitFlags to 1,
                                ; which means
                                ; "Don't send an EOL character
                                ; after printing the line".
;
;-- Find first non-blank char:
findspc cmpi.b  #' ',(argptr)+  ; Loop until nxt non-blank char found.
        ble.s   findspc
        subq.l  #1,argptr       ; When found, fix ptr &
        bra.s   setup           ; go to main scan loop.
;
;-- Reset ptr to start of line if there wasn't a '-n':
fixptr  move.l  argpstr,argptr  ; Reset argument line ptr
                                ; to start of line.
setup   clr.l   count           ; Clear output-chars count.
        clr.b   SpecFlag        ; Clear special chars flag.
;
;-- Make new output line from argument line:
scan    move.b  (argptr)+,char  ; Get first argument char (after
                                ; '-n', if there was a '-n').
        cmpi.b  #LF,char        ; Is it End-Of-Line char?
        beq     exit            ; If YES, then there's no more
                                ; argument line, & it's
                                ; time to show output.
;
;-- Check special chars flag:
        tst.b   SpecFlag        ; Check special chars flag.
        bne.s   doflag          ; If last char was special
                                ; one (either * or ^), go to doflag
                                ; to handle it.
;
;-- Check for quote:
        cmpi.b  #'"',char       ; If received char is quote,
        bne.s   chkspcl         ; avoid entirely.
        bchg    #1,BitFlags     ; NewEcho doesn't do quotes without
        bra.s   scan            ; '*' in front of them.
;
;-- Check for special chars (^ and *):
chkspcl cmpi.b  #'*',char       ; Is it '*'?
        bne.s   chkother
        btst    #1,BitFlags     ; If YES, was there a preceding '"'?
        beq.s   chkother        ; If NO, display it as a '*'.
        moveq   #1,SpecFlag     ; If YES, set flag to 1.
chkother cmpi.b #'^',char       ; Is it '^'?
        bne.s   ifflag
        moveq   #2,SpecFlag     ; If YES, set flag to 2.
ifflag  tst.b   SpecFlag        ; If SpecFlag is something other than 0
                                ; (meaning special char
                                ; was received),
        bne.s   scan            ; go back & get next char.
;
;-- Put char in buffer:
putinbuf move.b char,(bufptr)+  ; Put output char in output
                                ; buffer.
        addq.b  #1,count        ; Increment output-chars count by 1.
        bra.s   scan            ; Go up & get a new argument
                                ; line char.
;
;-- Handle special chars:
doflag  cmpi.b  #1,SpecFlag     ; If flag was set to 1 by last
                                ; char (meaning '*'),
                                ; this char is meant to be showed,
                                ; & needs no altering.
        beq.s   dfexit
        andi.b  #CTRLmask,d0    ; If it's 2 (meaning '^'),
                                ; knock this char down
                                ; to its equivalent CTRL char.
dfexit  clr.b   SpecFlag        ; In either case, clear special
                                ; chars flag,
        bra.s   putinbuf        ; & go up & store char in
                                ; output buffer.
```

```
;
;-- After looking at all argument chars:
exit        bclr.b  #1,BitFlags         ; Clear quotes toggle flag.
            tst.b   BitFlags            ; Check line termination type flag.
            bne.s   nolf                ; If it's 1, don't put a LF at end
                                        ;  of output string.
            move.b  #LF,(bufptr)        ; If it's 0, put a linefeed at end.
            addq.b  #1,count            ;  & increment count to handle it.
nolf        move.l  argpstr,d2          ;-Specify start of output line in d2;
            move.l  output,d1           ;  Specify outhandle in d1;
            move.l  count,d3            ;  Specify # of chars to write in d3;
            call    Write               ;  show the output.
;
;-- Close opened libraries:
close       move.l  dosbase,a1          ;-Specify Dos library in a1;
            move.l  AbsExecBase,a6      ;  then using Exec library,
            call    CloseLibrary        ;  close Dos library.
close1      rts                         ; Return control to CLI.
;
;-- Dos Library name:
dosname     dc.b    'dos.library',0  name of  Dos library
;-- Template string:
template    dc.b    ': '
endtemp

    end
```

Listing 2: Generates load file "Echo"

```
REM  "Echo.gen"
REM  AmigaBasic File Generator for "Echo".

CLS
COLOR 2:print "  *** ";
COLOR 3:PRINT "Echo Program Generator"
COLOR 2:print "  *** ";
COLOR 3:PRINT "Generated By MakeGen (Neal Bridges,
    02/87)."
COLOR 1:PRINT: PRINT "Please Wait, creating Echo. . ."
OPEN "Echo" FOR OUTPUT AS 1
PRINT
l=95: f=0
WHILE f=0
   Sum=0: l=l+5
   FOR i=1 TO 8
      READ x: IF x<>256 THEN
         IF x>=0 AND x<256 THEN PRINT #1,CHR$(x);
         Sum=(Sum+x) AND 255
      END IF
   NEXT :READ x:IF x<0 THEN READ x:f=1
   IF Sum<>x THEN
      PRINT "Error in Line";l
      CLOSE 1: KILL "Echo": END
   END IF
WEND
Done: CLOSE 1:PRINT "Echo Created."
END

100 DATA   0,   0,   3, 243,   0,   0,   0,   0, 246
105 DATA   0,   0,   0,   2,   0,   0,   0,   0,   2
110 DATA   0,   0,   0,   1,   0,   0,   0,  60,  61
115 DATA   0,   0,   0,   0,   0,   0,   3, 233, 236
120 DATA   0,   0,   0,  60,  42,  72,  38,  72,  28
125 DATA  40,   0,  67, 250,   0, 218,  66,  64, 193
130 DATA  44, 120,   0,   4,  78, 174, 254, 104,  10
135 DATA  42,   0, 103,   0,   0, 190,  44,  69, 192
140 DATA  78, 174, 255, 196,  46,   0,  12,   4, 253
145 DATA   0,   1, 111,   0,   0, 174,  12,  85, 127
150 DATA  63,  10, 102,  42,  69, 250,   0, 188, 212
155 DATA  36,  10,  34,   7,  38,  60,   0,   0, 185
160 DATA   0,   2,  44,  69,  78, 174, 255, 208,  62
165 DATA  78, 174, 255, 202,  34,   0,  36,  13,  24
170 DATA  38,  60,   0,   0,   0, 255,  78, 174,  93
175 DATA 255, 214,  40,   0,  96, 200,  66,   1, 104
180 DATA  36,  77,  12,  93,  45, 110, 102,  12, 231
185 DATA 114,   1,  12,  29,   0,  32, 111, 250,  37
190 DATA  83, 141,  96,   2,  42,  75,  66, 131, 124
195 DATA  66,   2,  16,  29,  12,   0,   0,  10, 135
200 DATA 103,   0,   0,  64,  74,   2, 102,  44, 133
205 DATA  12,   0,   0,  34, 102,   6,   8,  65, 227
210 DATA   0,   1,  96, 230,  12,   0,   0,  42, 125
215 DATA 102,   8,   8,   1,   0,   1, 103,   2, 225
220 DATA 116,   1,  12,   0,   0,  94, 102,   2,  71
225 DATA 116,   2,  74,   2, 102, 204,  20, 192, 200
230 DATA  82,   3,  96, 198,  12,   2,   0,   1, 138
235 DATA 103,   4,   2,   0,   0,  31,  66,   2, 208
240 DATA  96, 236,   8, 129,   0,   1,  74,   1,  33
245 DATA 102,   6,  20, 188,   0,  10,  82,   3, 155
250 DATA  36,  11,  34,   7,  38,   3,  78, 174, 125
255 DATA 255, 208,  34,  69,  44, 120,   0,   4, 222
260 DATA  78, 174, 254,  98,  78, 117, 100, 111, 242
265 DATA 115,  46, 108, 105,  98, 114,  97, 114,  29
270 DATA 121,   0,  58,  32,   0,   0,   3, 242, 200
275 DATA   0,   0,   3, 233,   0,   0,   0,   0, 236
280 DATA   0,   0,   3, 242, 256, 256, 256, 256,  -1
285 DATA 245
```

Listing 3: A Sample Command File

```
; "DateChange"
; A Command File by Neal Bridges, 02/87.
; A Short Routine to Change the System Date.
; Demonstration of new ECHO command.
;
FAILAT 25       ; prevent premature exit from sequence
; print blank line (^j), print 'The current. . .', no newline
ECHO -n ^j"The current date is "
DATE            ; print date
;
; print blank line, print 'Enter the. . .', change to colour #2
ECHO -n ^j"Enter the correct date: "^[[2m
; read date from keyboard
DATE >nil: ?
;
; change to colour 0, go up 3 lines, over 20 characters,
; and erase to End of Screen
ECHO -n ^[[0m^[[3A^[[20C^[[J
DATE            ; print new date where old date was
ECHO ""         ; print blank line
```

Programmed Cursor on the Amiga?

Jim Butterfield
Toronto, Ontario

. . .inside quotation marks, things work a little differently. . .

Slip your Amiga into CLI, and type the following command:

```
ECHO HOT**DOG
```

The computer will obediently echo HOT**DOG for you. You did know that ECHO's argument doesn't need those quotes, didn't you? You use quotes most of the time because what you want printed contains otherwise forbidden characters, mostly spaces.

That seems reasonable. Let's be more "conventional" and type:

```
ECHO "HOT**DOG"
```

Same thing, except for putting quote marks around the argument. But wait. . . one of the asterisks has disappeared! What's happening here?

It turns out that when you're inside quotation marks, things work a little differently. Especially asterisks: they work a lot differently. Without the quotation marks, asterisks are just like any other character. The quotation marks generate a special type of format. It's like the "quote mode" or "programmed cursor" you may have seen on earlier Commodore machines.

Special Treatment

When you type almost any CLI argument within quotation marks, the asterisk becomes something like an escape–type character. You might have seen something like this in C, or in programs such as PageSetter, except that a more usual character is the Backslash. It says: the next character to follow is not a regular character; instead, it's a special signal. Exception: when you give the character twice, the second character is accepted as typed. So the two asterisks collapse into one. If you like, you may translate the two characters as "Here's a special character. . . I just mean an ordinary asterisk".

The asterisk is a remnant of an Amiga system language called BCPL. This language was a forerunner of the C language, and you'll still find bits of it within the Amiga if you dig deep enough. Original BCPL had a considerable number of special codes generated by an asterisk; on the Amiga only four of them remain:

- `**` – generates a single asterisk;
- `*"` – generates quotation marks;
- `*n` – generates a newline (the "linefeed")
- `*e` – generates an ASCII escape character.

We've already tried the asterisk. You might try for some interesting effect with the next two by typing:

```
ECHO "PRESS *"RETURN*" WHEN READY"
ECHO "LINE 1*nLINE 2"
```

(Didn't know you could print two lines with one ECHO statement, did you?)

But the serious one is the *e sequence, which generates the ESC escape character. You might recognize this as CHR$(27) or hexadecimal 1B, but if you don't, no matter. It often has effects similar to those produced by pressing the ESC key. You probably know that pressing ESC followed by the "c" key clears the CLI screen. It also gets you out of certain annoying problems, such as a peculiar character set. Well then: If *e in quotes matches the ESC key, try this:

```
ECHO "*ec"
```

You were not surprised to see the screen clear, were you?

You might perhaps wonder what value this has. It's easier to press two keys, ESC and "c", than to type in the whole line shown above. But ECHO is often used in batch files (which I like to call EXECUTE files, because you usually invoke them by means of the EXECUTE command). So you can have a command sequence – not a program, just a command sequence – clear the screen. It can also do other things for you, which we'll get to in a moment.

File Nonsense

By the way, the use of the asterisk is not restricted to the ECHO command. Anything that you put in quotation marks uses asterisk–escape capabilities. So if you want to name a file SNOW*JOB you may either call it exactly that . . . or "SNOW**JOB". Remember: you only need to double the asterisk if you give the name within quotes.

You can also slip quotation marks into the file name itself. If you want to call a file "CAT" – not CAT, but "CAT", with the quotes as part of the name – you may do so by calling it "*"CAT*"". It will make it hard for others to reference the file by name. It will make it hard for YOU to reference the file by name. You really don't want to do this, except to prove that it can be done.

I'm happy to report that the DOS rejects fancier file names, the ones you'll try to create using the newline and escape sequences. After you think about it, you'll be glad it did.

The CSI

If you generate the ESC character (and we know how to do that, don't we?) you can follow it with a left–square–bracket character ("[") and the combination is called a CSI. CSI stands for Control

Sequence Introducer, which means that something special will follow. If you get into the machine, you can also create a CSI as a single character, CHR$(155) or hex 9B.

The CSI seems like a new form of ESCAPE, but it has quite a different character (no pun intended). The ESC is followed by one and only one action character; CSI may be followed by numeric data (which is noted) but terminates only when you reach the command character. The command character is usually alphabetic, and be careful: upper case versus lower case is significant here.

This business of number followed by command is rather like the system used in ED, the screen editor, for "extended" commands. Those are the commands you get by pressing the ESC key. Their system, as with CSI is: type the number first, and then the command letter; the command will be executed the appropriate number of times. CSI has its own commands, of course, not the ones that ED uses.

For example, the CSI command for "linefeed" is "E" (note the upper case). So you can command:

```
ECHO "*e[6EBANANA"
```

You'll be pleased to see the computer skip six lines and then print BANANA.

Just checking that you follow the syntax. In the above example, we MUST be in quotation marks; the asterisk indicates a special character follows; the e signals that this is an ESC sequence; the [says, "here comes a special command"; the 6 is read as the number of times to perform the following operation; the E is the command for linefeed, so we do six linefeeds; and, as I think Freud said, the BANANA is just a BANANA.

If you've coped with that one, I'll tell you that F (capitals again) is the command for NEGATIVE linefeed. So you might try:

```
ECHO "*e[5FHELLO"
```

You'll see something you might have considered impossible; the cursor going UP on the CLI screen! If you do much of this, you'll find yourself typing over old screen text, which is not fun. . . you still don't have true screen editing as you may know from earlier Commodore machines.

I won't go through all the curious commands that can be invoked with CSI. Check The AmigaDOS Manual (Bantam Books); within it, look at the appendix in The Developer's Manual. You'll find lots of commands to play with and make the CLI screen do strange things. There's one in particular I'd like to show you: command m (lower case), Set Graphic Rendition.

Graphic Rendition

The command, "CSI. . numeric. . m" changes the way things are shown on the CLI screen. The change is moderately permanent within the open CLI, so I try to change things back at the end of each line. So. . . you'll see two sequences of *e[within the quotes, one to produce the effect and one to get us back.

I like the "30" codes the best. They run from 30 to 33, and change the printing colour (30 is invisible, so is of less use). Try these:

```
ECHO "*e[32m DARK HORSE *e[31m"
ECHO "*e[33m RED ROVERS *e[31m"
```

You can change colours several times on a line, if you wish.

Now for the "40" codes. They change background colour as you print:

```
ECHO "*e[41m INVISIBLE *e[40m"
ECHO "*e[42m BLACK BAK *e[40m"
ECHO "*e[43m RED BACKS *e[40m"
```

You won't see the output from the first line very well, since it will be printed white on white. Note that the background colour only is changed here.

There are a few interesting numbers in the 0 to 7 range. Try:

```
ECHO "*e[1m Boldface! *e[0m"
ECHO "*e[3m Italics! *e[0m"
ECHO "*e[4m Underline *e[0m"
ECHO "*e[7m Reversed! *e[0m"
```

Note that code 0 brings the font back to normal.

A final note: you can use several of these codes, separated by semicolons. So as a last grand combination, try:

```
ECHO "*e[32;43mRead *e[3;4mThe Transactor
      *e[0;32;43m regularly!*e[0m"
```

You may have spotted that the sequence *e[0m restores everything to normal. When we used it in the middle of the last long line in order to cancel the italics and underline features, we had to reinstate the colours we were using.

Printer Footnote

The same CSI codes that work with the screen will often work with the printer. I often use ECHO with redirection in order to jot a note on the printer. Thus:

```
ECHO >PRT: " August 15/87 Listing"
```

This will spit out the correct message to the printer.

But it's interesting to see that many of the printer codes are similar to those we have used to the screen. The whole list of printer control codes may be found in the massive Amiga ROM Kernal Manual; it has also been partially reprinted in numerous places. A simple example will do here: if your printer has an italics font, type:

```
ECHO >PRT: "*e[3m Italics! *e[0m"
```

The same code that did the job to the screen will do it to the printer.

Conclusion

You can certainly pep up your Startup-Sequence file with a bit of colour and other special effects. And you might even have a better idea of some of the Amiga's marvellous inner space effects.

Adding Analog RGB Capability to the 1902 Monitor

Larry Phillips
Vancouver, BC

. . .analog RGB mode was present only on very early 1902 monitors. . .

When I first purchased my Amiga, I thought that I would be able to use a 1902 for the monitor, as it has a switch position for RGB analog. This feeling was reinforced by the fact that the Amiga's 1080 monitor looked identical to the 1902 in terms of the case, controls, and switches. Alas, it was not to be, for in Commodore's infinite wisdom, and probably at the insistence of the marketing types, the analog RGB mode was present only on very early 1902 monitors. Later versions had some components missing that crippled the analog RGB circuitry. This article describes how to add the components necessary. This modification will work on the 1902 only, and NOT on the 1902A.

Note: If you do not feel comfortable working inside your monitor, this modification is not for you! If this is the case, either find someone who has the experience and knowledge necessary or take this article and your 1902 to a qualified service technician. Read this disclaimer until you realize that I will not be held responsible for any damage you may do to yourself or your monitor.

If you find the components already in place, yet the monitor will not allow analog RGB, it is possible that some traces have been cut on the board near the RGB switch. If so, you are on your own, but simply reconnecting them should be sufficient to get it working.

Now to the modification itself. Figure 1 is the complete parts list, and all components are both commonly available and low cost. Figure 2 shows the layout of the board and the approximate location of the parts to add. All parts to be added are clearly marked with silk screen on the 1902 main board. Step by step, here we go:

1. Remove the back cover
   - 6 screws
2. Remove shield on bottom of main board
   - unsolder and twist tabs
3. Remove metal mounting plates holding input jacks and small boards.
   - 3 screws through main board
   - 1 screw holding the two plates together
   - 1 screw holding RCA jack block
   - 2 posts holding RGB connector
4. Remove metal shield from component side of board.
   - twist tabs
5. Locate and suck the solder from the holes for the components to be added.
   - R7267/8/9 in area 1, near IC7201
   - R7270/1/2 in area 3
   - D7218/19/20 in area 3
   - C7202/3/4 in area 3
   - C7219/20 in area 2
   - JM123/4/5 in area 3
6. Solder in components, paying particular attention to the polarity of the electrolytic capacitors and diodes. Capacitors are marked on the board's silk screening with a 'dot' near the negative lead. Diodes are shown with a picture just like the schematic representation.

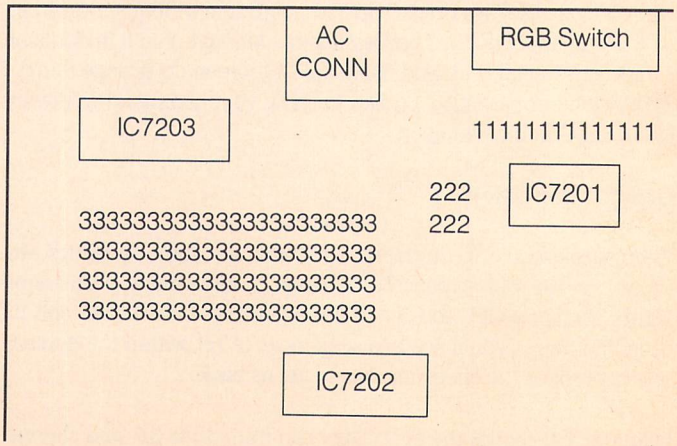
NOTE: the board is single sided, and the bonding is not the best on the traces, so use caution and low heat so you won't damage the traces.

7. Replace the top shield and metal mounting plates.
8. Test your work at this time by placing the switch in the third position (to the right as seen from the rear of the monitor, facing the switch) and hooking the Amiga to the monitor through a standard Amiga RGB cable.
9. If all is well, replace the bottom shield and resolder the tabs
10. Some 1902 monitors had a smaller hole through which the RGB switch protruded, allowing it to be used in only 2 positions (positive and negative RGBI). If this is the case, cut the slot in the case to allow the switch to be placed in the newly acquired RGB analog position.
11. Replace the cover. Enjoy!

Figure 1: Parts List

Resistors		Capacitors		Diodes	
R7267	150 ohms	C7202	330 uf – 10V	D7218	1N4148
R7268	150 ohms	C7203	330 uf – 10V	D7219	1N4148
R7269	150 ohms	C7204	330 uf – 10V	D7220	1N4148
R7270	820 ohms	C7219	.01uf	Jumpers	
R7271	820 ohms	C7220	47 uf – 10V	JM123	wire
R7272	820 ohms			JM124	wire
				JM125	wire

Figure 2: As viewed from bottom of board



Amiga Dispatches

by Tim Grantham, Toronto, Ontario

As I write this, the stock price of CBM has dropped yet again to about 8.50 on the NYSE, where it has steadied. A quarterly report is due out in the middle of August and I would not be at all surprised if it contains news of a loss, news that has already leaked to Wall Street.

We will all know by the time you read this, of course, but don't be too disheartened if it does. CBM does not seem to be laying off any more staff. In fact, they seem to be hiring on a small basis. In addition, part of the loss may be attributable to initial manufacturing costs of the 500 and the 2000. Reports indicate that the 500 has gotten off to a good start in sales and that should be reflected in the next quarterly report.

I have recently joined two more information services in my obsessive quest for Amiga information: GEnie and BIX. The latter is pretty expensive -- accessed from Canada through Tymnet, it is $15 (US) per hour at 1200 bps. That figure is for non-prime hours and there is an additional $25 (US) sign-up charge.

I found it particularly irritating that nowhere are potential Canadian customers of BIX told that the Tymnet access charge is $6 (US) rather than the $2 (US) spelled out in the ads in *BYTE* and in the sign-on message. Most Canadians don't find out about the extra charge until they get their bills, I imagine. The only reason *I* found out was because my nasty suspicious nature made me ask the service rep who had called to confirm my sign-on. He had to go to some lengths to find out himself. *Grrrr.*

BIX is a great place for in-depth Amiga technical information, though. I was able to get help on Charlie Heath's **getfile** requester for my **keep** program (coming RSN) directly from cheath himself. To many professional programmers, the cost of the service is well worth the information gained.

GEnie, on the other hand, is clearly intended to be a consumer network. So far, they have been a model of efficiency and service: costs are clearly stated, in Canadian dollars for Canadian customers, and documentation arrived promptly. There is a vigorous Amiga club, *Starship Amiga*, operated by Deb Christensen. deb! defected from the Delphi network, where she had arrived after CompuServe had booted CBM from operation of its Commodore forums. I have not had an opportunity to spend much time wandering the decks of *Starship Amiga*, but for less than $10 (Can.) per hour (plus the $25 (Can.) signup fee), it appears to well worth investigating.

This brings my network memberships to five. However, it has been so long since I last accessed Delphi that I have misplaced my password. If you think that is indicative of my opinion of the value of Delphi, I shan't correct you.

PD Tips

As usual, the arena (or should I say 'free-for-all'?) of public domain software has seen the arrival of some powerful new contenders. **ACO 2.00** by Steve Pietrowicz, currently available exclusively on PeopleLink, is an Amiga version of a concept originally introduced on a Macintosh program entitled **VMCO**. Faithful readers of this column may remember my description of VMCO in the late lamented *TPUG Magazine*.

The program is intended for live telecommunicating. The display consists of a series of empty chairs around a conference table. As each attendee arrives, the empty chair is replaced with a picture of the new arrival. When that person makes a comment, they can switch the picture displayed to reflect the content of the comment. One might have a happy face, a 'flame' or angry face, a 'back-in-a-minute' face, and so on. Of course, this display only appears on the machines of those attendees who are running the software, and have the faces of each attendee stored in the program's face libraries.

The Amiga version, ACO, adds colour to the proceedings, so to speak. **DPaint** can be used to create your own faces, or they can be designed with the **ACOfaces** program available as part of the ACO.arc file. The whole thing is built on Dan James' excellent **comm** terminal program.

Journal is an intriguing program. It records all InputEvents generated by the input.device, including timings, and stores them into a file. These events consist of key presses and mouse actions. Then, using the accompanying **Playback** program, the sequence of InputEvents can be replayed through the input.device, fooling the Amiga into thinking they are really originating from the operator. The result is an exact playback of the user's interaction with the computer. Talk about the ghost in the machine!

Journal suggests a number of interesting applications: as a way to create script files for terminal programs; for software tutorials; and for product demonstrations. Users of DPaint II, for example, could see just how Jim Sachs created all those wonderful graphics.

Two new languages have arrived on the scene: **Draco** and **Icon**.

Draco is described by its inventor, Chris Gray of Edmonton, Alberta, as a language combining all the best features of C and Pascal. Originally developed for CP/M-80 systems, Jeffries has added a Draco compiler for the Amiga.

Icon is a descendant of SNOBOL, an interesting language with a host of powerful string-handling features. Icon also offers list processing and set manipulation commands, but is better integrated and more consistent than Snobol. Scott Ballantyne has ported version 6 of Icon to the Amiga already, and is now porting 6.3, which should be ready this fall.

Several issues ago, I hinted broadly that CBM should make the include files available to the general public at a nominal charge, so that people like Jim Butterfield could leave AmigaBASIC behind and start digging through the OS in assembly language and C without paying an arm and a leg for a compiler. I'm pleased to announce that CBM is doing exactly that: for $20 (US) you get two disks containing the C and assembly language include files and the 1.2 Autodocs to tell you what it's all about. If you want to order them, send a request and a cheque to Laurie Brown at CBM's West Chester HQ.

A note of warning: currently, the only PD assembler I know of that is completely compatible with the .i include files, is **A68k** by Charlie Gibbs. Others require that the include files be reformatted first.

TAB Books has published *Amiga Assembly Language Programming*, (#2711-H, $19.95). I don't know if it's any good, but it is the first of its kind.

Hard and soft news

You may remember my complaints last issue about **Amiga Tax**, a Canadian income tax calculation program. David Sopuch, one of the programmers, tells me that registered owners will be notified of the availability of an upgrade that fixes bugs and adds enhancements. Beta testers and Revenue Canada willing, it should be ready by December, at a cost of $29.95.

Toronto's Eric Haberfellner has completed the latest version of his fine shareware VT100/102 terminal emulator **Handshake**. He has added Ymodem and batch Ymodem file transfers, and fixed some minor bugs. In a survey of available VT100 emulators recently carried out on Usenet, Haberfellner's program came out on top. All he requests is $25, folks.

Comspec is delivering their version of a recoverable RAM disk (RRD) to owners of the AX2000 memory boards. The first 900 or so AX2000 units require a minor hardware fix to use the RRD. This is provided as part of the upgrade package, in addition to the software. Comspec's RRD is slightly different from others in that the Amiga literally sees it as another disk drive – it can be formatted, diskdoctored, disksalved and diskcopied. Most importantly, of course, it can survive a warm reboot (ctrl-amiga-amiga).

By the time you read this, Comspec will have also announced the availability of their SCSI hard drives. Domenic DeFrancesco, designer of the both the RAM board and the SCSI interface, told me that the SCSI interface will autoconfigure, has a bus pass-through and does not interfere with multitasking in any way. It will provide transfer rates of nearly 290 Kb per second for reads and 250 Kb for writes, once the AmigaDOS hard drive support arrives from CBM. The drive chassis has room for one full height drive, or 2 half-height drives. In addition, two ports are provided to chain several other SCSI devices, including Macintosh hard drives. First offerings consist of a 20 Mb and a 40 Mb drive. The SCSI interface itself will likely be available separately in the near future.

ASDG Inc. makes a number of products that have been very well received, including **Facc**, a floppy disk access accelerator program, and a 2 Mb Zorro RAM board that is upgradable to 8 Mb. Now they are working on two other products: the **2000-and-1** and **SDP** (Satellite Disk Processor). The former, to cost about $800 (US), is a box for the 1000 that will accept plug-in cards intended for both the 1000 and the 2000, including the Bridge card. The SDP, as I understand it, will reorganize the tracks on a disk to provide faster access. All ASDG hardware comes with an unusually long 18-month warranty; all software is provided with lifetime free upgrades, and is not copy protected. *Bravo!*

Michigan Software, the makers of the Insider RAM board, are now selling **Kwikstart**. This kit provides Kickstart 1.2 in ROM for the 1000, as it is now for the 500 and 2000. It doesn't lock you out of future versions of Kickstart, however. By simply holding the warm reboot keys a little longer, the Kwikstart ROMs will be switched out, and you can boot with a different Kickstart. The product works with the Insider board and costs $169.95 (US). Using the Kwikstart ROMs also frees 256K of the RAM previously used by the disk-loaded Kickstart – a well thought out product.

VideoScape 3D is now available and the source of some spectacular 3D animation demos. Look for them on the new series of CBM television ads to start running in September. . . **Digiview 2.0** and **Digipaint 2.0** have also arrived. The former provides some first-rate image processing software and the latter is a HAM-mode paint program that permits drawing with all 4096 colours. . . The makers of **Acquisition** and **Datamat**, two powerful relational database management programs, have responded to vociferous complaints of difficulty of use with improved user interfaces and documentation. . . Mimetics, publishers of the **Soundscape** system, have expanded their developer support to include full documentation of all the **soundscape.library** functions and Aztec C support. . .

Online 2.00 is a significant upgrade to the original with VT100 and Tektronics terminal emulations added; Kermit, Ymodem and Zmodem protocols supported; autochop; ANSI colour; and the ability to keep up with text output. It no longer hogs the printer device, either. However, a small bug has surfaced: if you disable the autochop feature, Xmodem transfers will no longer work. Publisher Micro Systems Software is providing a free fix.

Diga! (wish I knew what the name meant), from Aegis Development, appears to be following in the footsteps of Soundscape. The company is posting documentation on the terminal emulations and the proprietary DoubleTalk protocol on the networks to encourage outside developers. The new version of the program being developed by Aegis will include the WXmodem protocol used by PeopleLink, and a chat window for live conferencing.

By now you may know that the 500 provides NTSC output in monochrome only. Those who need a colour output for their VCR have three possible solutions. Mimetics will be selling genlocks for all three Amigas that, in addition to synchronizing the Amiga output to another video source, provide a colour NTSC signal from the RGBA output. CBM will be selling an an encoder that does the same thing for about $50 (US). Or you can make your own encoder, using the same Motorola 1377P chip that the 1000 uses, for about $25 in parts. You can write to Motorola for free documentation that includes a schematic.

The Commodore 64 emulator I have mentioned in the past should be out by the time you read this. It will cost about $150 (US), and consists of both software and a hardware interface that plugs into the parallel port and permits connection of the 1541 disk drive. The makers claim emulation is complete, with the exception of the SID chip, and they now have GEOS running on the Amiga. The emulator takes over the computer, however. No multitasking possible.

Microfiche Filer provides a uniquely Amiga way of viewing a database: as the name implies, all the records are present on the screen in a miniaturized format. One can scan them with a 'magnifying glass' and zoom in on particular records. Because it handles graphic data as well as text and numbers, this is an ideal means to store and display real-estate properties, personnel records and so on. A possible limitation is that the entire database must be in memory at once.

Darrin Massena's **MenuEd** program now works under 1.2. Now, if only someone would finish the **Egad** gadget editor. . . **Textcraft Plus** has been released in Europe. No word about when it will cross the sea. . . SoftCircuits Inc. has announced that a schematic capture program called **Scheme** is in beta test and will soon be available for $499.95 (US). It is compatible with their successful **PCLO** printed circuit board layout software series, which now includes **PCLOjr**, a scaled-down version for home use selling at $69.95 (US). . . The German version of the 2000, available in Canada and Europe, has wait states in the RAM plugged into the so-called CPU bus slot that slow the machine to about 85-90% of the speed of a 1000. The 500 and the West Chester 2000 (when it arrives) will not have this problem.

Joanne Park here in Toronto removed a resistor from the encoder circuit in her 1000 as described in *Amazing Computing*, Vol. 2, No. 7, tested it with a vectorscope and found that the colour balance was bang on. However, her Amiga was a pre-June '86 machine: the *Amazing* article said the removed resistor in these machines should be replaced with a 470K Ohm resistor. I suppose the lesson here is that those who need to make the change should get someone with a vectorscope to check it. . .

TDI will get some needed competition when Oxxi, of Maxiplan reknown, comes out with their Modula 2 compiler, called **Benchmark**. Reports from beta testers indicate that it is very fast. . . **City Desk**, a desktop publishing program written for MicroSearch Inc. by SunRize Industries (who also make the **PerfectSound** stereo digitizer), is now available. For more info, contact Tom Hayden at (713) 988-2818, 9-6 CST. . . Two attractive disk-based magazines have appeared for the Amiga, *The New Alladin* and *AMnews*. The latter is of particular interest to Canadians since it is published in Montreal. The full address is Vertex Associates, 415 Trenton Ave., Montreal, PQ, H3P 2A1. Phone: (514) 739-3301.

Has anyone besides me noticed the bug in the **opt i** version of the **dir** command that appears when it is used with the ram: disk? If you delete a file, the file disappears but the memory does not get deallocated. You can have an empty ram: disk that takes up 200K of precious memory!

I'm glad to see that an increasing number of third-party Amiga developers have established a network presence. Aegis Development and MicroIllusions have their own conferences (forums) on BIX; Micro Systems Software fields questions on CIS; ASDG, C Ltd., and Gold Disk make regular appearances on major networks. Most Amiga developers have proven to be accessible and responsive to their customers and I hope they continue to do so.

The Amiga, network computing and the whole damn thing

In my nine-to-five capacity as assistant editor of Design Engineering magazine, I have been able to gain a fairly broad perspective on the use of computers, from micros to supercomputers.

One of the most exciting developments is in the area of 'distributed computing'. This requires multiple computers communicating over a network. The idea is to distribute the execution of a single large program over all the machines in the network, assigning procedures to the most appropriate computer. Image processing functions, for example, could be executed by a dedicated array processor on the network, while the user interface for the program could be handled by a personal computer. The entire process would be transparent to the user, sitting at his or her console. More importantly, it would be independent of the network, computers or operating systems being used – a tall order, but one that embodies the dream of integrating all computer resources into one huge computing system.

Xerox developed the first distributed computing system, implementing RPCs (remote procedure calls) under its XNS (Xerox Network System). Sun Microsystems made RPCs more broadly available by implementing them under TCP/IP, an industry standard networking protocol, and NFS (Network File System), its own network data-sharing protocol that has become a de facto industry standard. Now Apollo has thrown its hat into the token ring by placing the specifications for its NCS (Network Computing System) into the public domain.

Apollo's scheme is the most flexible, powerful system yet, for a number of reasons. One, in addition to being machine and operating system independent, it's network independent. It could quite literally be implemented over a telephone line with a modem at each end, though the transmission overhead would slow down program execution probably beyond the point of practicality. Two, it supports concurrent programming. The same procedure could be installed and called on multiple machines simultaneously, with all results funnelled to the calling program.

(I recently saw a demonstration of over 30 Apollo workstations concurrently calculating a Mandelbrot picture. What would normally have taken well over two hours on one workstation took *seven minutes* on the network. Bear in mind that all the workstations were simultaneously being used by their operators for other tasks – some were not even aware that their machines were also working on the Mandelbrot picture.)

Thirdly, it provides for dynamic load balancing across the network. What this means is that if one computer has more CPU time free than another, the system will have the RPC executed by that machine. Besides providing more efficient use of the computing hardware available, this makes life easier for the programmer writing the distributed application. He or she does not have to know which machine(s) carry the code for the RPC.

Unfortunately for Apollo, the industry momentum is behind Sun's NFS system. Adopting Apollo's superior system would mean a good deal of retrenching. Nevertheless, people are impressed with the system's features. Apollo has established the Network Computing Forum to discuss distributed computing and over thirty companies have joined, including Sun, Apple, Texas Instruments and Motorola. My sources indicate that members want to see the features of NCS developed under NFS.

Where does the Amiga fit into this? Now that Ameristar Technologies has developed Ethernet boards for the Amiga and NFS client software, the Amiga can fit into a distributed computing environment. It would provide much of the same functionality as the workstations – the graphic interface, the multitasking – but at a low cost. With the concurrent computing capability provided by NCS, a network consisting entirely of Amigas could simultaneously execute a single program.

The more I learn about the industry as a whole, the more I see the Amiga filling a large niche between the workstation and the micro. IBM and Apple have each developed such products but at several times the price of the Amiga without several times the capability.

I'm always interested in any comments or questions you may have. They can be sent to me c/o The Transactor, or via electronic mail addressed to 71425,1646 on CompuServe; AMTAG on PeopleLink; dispatcher on BIX; t.grantham on GEnie; or to Tim Grantham on Wayne Beyea's fine Bloom Beacon BBS at (416) 297-5607.

News BRK

Submitting NEWS BRK Press Releases

If you have a press release you would like to submit for the NEWS BRK column, make sure that the computer or device for which the product is intended is prominently noted. We receive hundreds of press releases for each issue, and ones whose intended readership is not clear must unfortunately go straight to the trash bin. It should also be mentioned here that we only print product releases which are in some way applicable to Commodore equipment. News of events such as computer shows should be received at least 6 months in advance.

Transactor News

No-Fault Program Entry Insurance

We get several letters in the wake of each issue reporting problems with programs typed in from listings in the magazine. We understand the frustration readers feel when they have typed in several pages of DATA statements or assembler source only to find that the resulting program runs either incorrectly or not at all, whether it was their error or ours that is to blame.

We're hoping that our new No-Fault Program Insurance system will provide a way out. Here's how it works: let's suppose you've just entered a long program from the latest issue and saved it to disk. Now you run it but your machine crashes, so you unload a few appropriate expletives and load it back in to recheck the Verifizer codes. A few hours later, your eyeballs are spinning in counter-rotating circles and you still haven't found the bug, so you save the program to a new disk, label it clearly with your name, address and phone number, plus the volume and issue number you were working from, and drop it in the mail to us with a brief description of the problem.

When we get your disk, we'll return it with the official version of the program alongside your own as soon as possible. You'll be out the price of the postage, but you'll have a working program. We'll be out the postage too, but we'll have a hand-entered copy of the program with which we can find out whether the listing in the magazine contains an error. That will make it a lot easier for us to track down the occasional bug amongst the many programs we publish, so that we can print corrections as soon as possible after the original version appears.

By the way, if you'd like to check with us by phone for bug reports before you mail in the disk, that's fine, and just might save you the extra trouble of getting the package ready.

Send TPUG Subscriptions to TPUG!

If you wish to receive the Transactor containing the insert for TPUG members (Toronto Pet Users Group), you must be a TPUG member. To become a TPUG member, write them at 5300 Yonge Street, Toronto, Ontario, M2M 5R2.

You can do yourself, TPUG, and us a favour by writing to them directly. Using our subscription cards to start or renew TPUG memberships may save you a few cents in postage, but can lead to bureaucratic slip-ups resulting in shipment of the wrong magazine, or at best a delay in your subscription. Also, please remember to make cheques payable to "TPUG Inc." for TPUG memberships.

Transactor Renewals

Many subscription renewals arrive with a request that they be "back-dated" to a previous issue. The main reason we can't oblige is the postage difference between copies that go out as part of the bulk subscriber mailing and a single copy at regular rates (you may have noticed that our price for back issues includes the cost of postage). Subscriptions can only be started with the next issue due out. Earlier magazines must be ordered as back issues.

Subscription orders must be received at least three weeks before the release of an issue to ensure that the subscription will start with that issue. So, if you order a subscription and don't get the next issue that comes out, it may be that we received it too late to get your name to the printer; your first issue should be the following one.

Use the New Subscription Cards!

When ordering subscriptions or Transactor products, please use a subscription card from the most recent issue. The cards are constantly being updated, and product and price information from older cards is no longer valid. Sub cards that show our old address will take longer to reach us, and many of the products listed are no longer available. The most notable difference between the old and new sub cards is the Canadian price for a year's subscription: it is now $19.00, and the $15.00 price listed on the older cards will no longer be honoured.

New Transactor Special Offers

Beginning with this issue, we are starting a policy of special offers every issue. The subscription card announces the offer, which is valid for items ordered from that subscription card only, and only for the specified time.

The special offer this issue is for Transactor disks: buy any three, and get two free! This does not apply to disk subscriptions, only to three disks ordered at one time. Simply use the order card (insert from this issue only), mark off any 5 disks, but send payment for only 3 (or check off 10 and pay for 6). This offer applies only to orders postmarked before November 1, 1987, so we recommend you take advantage of it now!

New Combo-Subscription Premiums

Up to now, when you ordered a combined magazine/disk subscription, we sent you a free Transactor T-Shirt to sweeten the deal. Sorry, but we have to withdraw that offer, effective immediately. Instead, you now get your choice of the T-Shirt, the Potpourri Disk, or the TransBasic 2 Disk and Manual – whichever suits you best! Just check the appropriate box at the top of the Order Form when you fill in your combo-subscription application.

U.S. Orders Invoiced In Canadian Dollars

If you ordered a subscription from the U.S. and received an invoice for $19.50, don't be alarmed: that's 19.50 Canadian dollars, which works out to just under $15.00 US at current exchange rates.

Advertisers Wanted

If anyone is interested in placing full-page, half-page or quarter-page colour or black and white ads in the Transactor, please contact us for rates and information. Yup, you heard right. We'll take ads now, but space is limited. Our ceiling currently is the cover spots plus 5 pages of the interior.

Group Subscription Rates: The 20/20 Deal

The Transactor has always been popular among Commodore user groups, so to encourage new subscribers we are offering quantity discounts for magazine and disk subscriptions: 20 percent off for group orders of 20 or more subscriptions. If you can get together enough friends or club members, just put all the subscriptions in a single envelope, and you get the discount. You don't need to be a user group to qualify – any 20 or more subscription cards in a single package get the 20/20 deal, no questions asked.

Mail-Order Products No Longer Offered

We have removed several products from our mail-order card: The Gnome Speed Compiler and Gnome Kit Utility, the "pocket" series of software, PRISM's SuperKit 1541, the BH100 hardware course material, the Anchor Volksmodems, and the Comspec 2 megabyte RAM expansion units. We still have some stock of the software and can order more of the other products if necessary, so we should be able to fill any orders from previous subscription cards.

Transactor Mail Order

The following details are for products listed on the mail order card. If you have a particular question about an item that isn't answered here, please write or call. We'll get back to you and most likely incorporate the answer into future editions of these descriptions so that others might benefit from your enquiry.

■ Moving Pictures - the C-64 Animation System, $29.95 (US/C)
This package is a fast, smooth, full-screen animator for the Commodore 64, written by AHA! (Acme Heuristic Applications!). With Moving Pictures you use your favourite graphics tool to draw the frames of your movie, then show it at full animation speed with a single command. Movie 'scripts' written in BASIC can use the Moving Pictures command set to provide complete control of animated creations. BASIC is still available for editing scripts or executing programs even while a movie is being displayed. Animation sequences can easily be added to BASIC programs. Moving Pictures features include: split screen operation – part graphics, part text – even while a movie is running; repeat, stop at any frame, change position and colours, vary display speed, etc; hold several movies in memory and switch instantly from one movie to another; instant, on-line help available at the touch of a key; no copy protection used on disk.

■ The Potpourri Disk, $17.95 US, $19.95 Cdn.
This is a C-64 product from the software company called AHA!, otherwise known as Nick Sullivan and Chris Zamara. The Potpourri disk is a wide assortment of 18 programs ranging from games to educational programs to utilities. All programs can be accessed from a main menu or loaded separately. No copy protection is used on the disk, so you can copy the programs you want to your other disks for easy access. Built–in help is available from any program at any time with the touch of a key, so you never need to pick up a manual or exit a program to learn how to use it. Many of the programs on the disk are of a high enough quality that they could be released on their own, but you get all 18 on the Potpourri disk for just $17.95 US / $19.95 Canadian. See the Ad in this issue for more information.

■ TransBASIC II $17.95 US, $19.95 Cdn.
An updated TransBASIC disk is now available, containing all TB modules ever printed. The first TransBASIC disk was released just as we published TransBASIC Column #9 so the modules from columns 10, 11 and 12 did not exist. The new manual contains everything in the original, plus all the docs for the extras. There are over 140 commands at your disposal. You pick the ones you want to use, and in any combination! It's so simple that a summary of instructions fits right on the disk label. The manual describes each of the commands, plus how to write your own commands.

People who ordered TB I can upgrade to TB II for the price of a regular Transactor Disk (8.95/9.95). If you are upgrading, you don't necessarily need to send us your old TB disk; if you ordered it from us, we will have your name on file and will send you TB II for the upgrade price. Please indicate on the order form that you have the original TB and want it upgraded.

Some TBs were sold at shows, etc, and they won't be recorded in our database. If that's the case, just send us anything you feel is proof enough (e.g. photocopy your receipt, your manual cover, or even the diskette), and TB II is yours for the upgrade price.

■ The Amiga Disk, $12.95 US, $14.95 Cdn.
Finally, the first Transactor Amiga disk is available. It contains all of the Amiga programs presented in the magazine, of course, including source code and documentation. You will find the popular "PopColours" program, the programmer's companion "Structure Browser", the Guru–killing "TrapSnapper", user–friendly "PopToFront", and others. In addition, we have included public domain programs – again, with documentation – that we think Transactor readers will find useful. Among these are the indispensable ARC; Csh, a powerful CLI–replacement DOS shell; BLink, a linker that is much faster and has more features than the standard ALink; Foxy and Lynx, a 6502 cross assembler and linker that makes its debut on the Amiga Disk; and an excellent shareware text editor called UEdit. In addition, we have included our own expression–evaluator calculator that uses variables and works in any number base. All programs contain source code and documentation; all can be run from the CLI, and some from Workbench. There's something for everyone on the Transactor Amiga disk.

■ Transactor T-Shirts, $13.95 US, $15.95 Cdn.
■ Jumbo T-Shirt, $17.95 US, $19.95 Cdn.
As mentioned earlier, they come in Small, Medium, Large, Extra Large, and Jumbo. The Jumbo makes a good night-shirt/beach-top – it's BIG. I'm 6 foot tall, and weigh in at a slim 150 pounds – the Small fits me tight, but that's how I like them. If you don't, we suggest you order them 1 size over what you usually buy.

One of the free gift choices we offer when you order a combination magazine AND disk subscription is a Transactor T-Shirt in the size and colour of your choice (sorry, Jumbo excluded). The shirts come in red or light blue with a 3-colour screen on the front featuring our mascot, Duke, in a snappy white tux and top hat, standing behind our logo in 3D letters.

■ Inner Space Anthology $14.95 US, $17.95 Cdn.
This is our ever popular Complete Commodore Inner Space Anthology. Even after two years, we still get inquiries about its contents. Briefly, The Anthology is a reference book – it has no "reading" material (ie. "paragraphs"). In 122 compact pages, there are memory maps for 5 CBM computers, 3 Disk Drives, and maps of COMAL; summaries of BASIC commands, Assembler and MLM commands, and Wordprocessor and Spreadsheet commands. Machine Language codes and modes are summarized, as well as entry points to ROM routines. There are sections on Music, Graphics, Network and BBS phone numbers, Computer Clubs, Hardware, unit-to-unit conversions, plus much more. . . about 2.5 million characters total!

■ The Transactor Book of Bits and Pieces #1, $14.95 US, $17.95 Cdn.
Not counting the Table of Contents, the Index, and title pages, it's 246 pages of Bits and Pieces from issues of The Transactor, Volumes 4 through 6. Even if you have all those issues, it makes a handy reference – no more flipping through magazines for that one bit that you just know is somewhere. . . Also, each item is forward/reverse referenced. Occasionally the items in the Bits column appeared as updates to previous bits. Bits that were similar in nature are also cross-referenced. And the index makes it even easier to find those quick facts that eliminate a lot of wheel re-inventing.

■ The Bits and Pieces Disk, $8.95 US, 9.95 Cdn.
■ Bits Book AND Disk, $19.95 US, 24.95 Cdn.
This disk contains all of the programs from the Transactor book of Bits and Pieces (the "bits book"), which in turn come from the "Bits and Pieces" section of past issues of the magazine. The "bits disk" can save you a lot of typing, and in conjunction with the bits book and its comprehensive index can yield a quick solution to many a programming problem.

■ The G-LINK Interface, $59.95 US, 69.95 Cdn.
The Glink is a Commodore 64 to IEEE interface. It allows the 64 to use IEEE peripherals such as the 4040, 8050, 9090, 9060, 2031, and SFD-1001 disk drives, or any IEEE printer, modem, or even some Hewlett-Packard and Tektronics equipment like oscilloscopes and spectrum analyzers. The beauty of the Glink is its "transparency" to the C64 operating system. Some IEEE interfaces for the 64 add BASIC 4.0 commands and other things to the system that sometimes interfere with utilities you might like to install. The Glink adds nothing! In fact it's so transparent that a switch is used to toggle between serial and IEEE modes, not a linked-in command like some of the others. Switching from one bus to the other is also possible with a small software routine as described in the documentation.

As of Transactor Disk #19, a modified version of Jim Butterfield's "COPY–ALL" will be on every disk. It allows file copying from serial to IEEE drives, or vice versa.

■ The Tr@ns@ctor 1541 ROM Upgrades, $59.95 US, $69.95 Cdn.
You can burn your own using the ROM dump file on Transactor Disk #13, or you can get a set from us. There are 2 ROMs per set, and they fix not only the SAVE@ bug, but a number of other bugs too (as described in P.A. Slaymaker's article, Vol 7, Issue 02). Remember, if SAVE@ is about to fail on you, then Scratch and Save may just clobber you too. This hasn't been proven 100%, but these ROMs will eliminate any possibilities short of deliberately causing them (ie. allocating or opening direct access buffers before the Save).

NOTE: Our ROM upgrade kit does NOT fit in the 1541C drives. Where we supply two ROMs, Commodore now has it down to one MASSIVE 16 Kbyte ROM. We

don't know if the new drives still contain the bugs eliminated by our kit, but we'll find out and re-cut a second kit if necessary. In the meantime, 1541C owners should not order this item until further notice.

■ The Micro Sleuth: C64/1541 Test Cartridge, $99.95 US, $129.95 Cdn.
We never expected this cartridge, designed by Brian Steele (a service technician for several schools in southern Ontario), would turn out to be so popular. The Micro Sleuth will test the RAM of a C64 even if the machine is too sick to run a program! The cartridge takes complete control of the machine. It tests all RAM in one mode, all ROM in another mode, and puts up a menu with the following choices:

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

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

Transactor Disks, Transactor Back Issues, and Microfiche

All Transactors since Volume 4 Issue 01 are now available on microfiche. According to Computrex, our fiche manufacturer, the strips are the "popular 98 page size", so they should be compatible with every fiche reader. Some issues are ONLY available on microfiche – these are marked "MF only". The other issues are available in both paper and fiche. Don't check both boxes for these unless you want both the paper version AND the microfiche slice for the same issue.

To keep things simple, the price of Transactor Microfiche is the same as magazines, both for single copies and subscriptions, with one exception: a complete set of 24 (Volumes 4, 5, 6, and 7) will cost just $49.95 US, $59.95 Cdn.

This list also shows the "themes" of each issue. Theme issues didn't start until Volume 5, Issue 01. Transactor Disk #1 contains all programs from Volume 4, and Disk #2 contains all programs from Volume 5, Issues 1-3. Afterwards there is a separate disk for each issue. Disk 8 from The Languages Issue contains COMAL 0.14, a soft-loaded, slightly scaled-down version of the COMAL 2.0 cartridge. And Volume 6, Issue 05 lists the directories for Transactor Disks 1 to 9.

■ Vol. 4, Issue 01 (■ Disk 1)
■ Vol. 4, Issue 02 (■ Disk 1)
■ Vol. 4, Issue 03 (■ Disk 1)
■ Vol. 4, Issue 04 – MF only (■ Disk 1)
■ Vol. 4, Issue 05 – MF only (■ Disk 1)
■ Vol. 4, Issue 06 – MF only (■ Disk 1)
■ Vol. 5, Issue 01 – Sound and Graphics (■ Disk 2)
■ Vol. 5, Issue 02 – Transition to Machine Language – MF only (■ Disk 2)
■ Vol. 5, Issue 03 – Piracy and Protection – MF only (■ Disk 2)
■ Vol. 5, Issue 04 – Business & Education – MF only (■ Disk 3)
■ Vol. 5, Issue 05 – Hardware & Peripherals (■ Disk 4)
■ Vol. 5, Issue 06 – Aids & Utilities (■ Disk 5)
■ Vol. 6, Issue 01 – More Aids & Utilities (■ Disk 6)
■ Vol. 6, Issue 02 – Networking & Communications (■ Disk 7)
■ Vol. 6, Issue 03 – The Languages (■ Disk 8)
■ Vol. 6, Issue 04 – Implementing The Sciences (■ Disk 9)
■ Vol. 6, Issue 05 – Hardware & Software Interfacing (■ Disk 10)
■ Vol. 6, Issue 06 – Real Life Applications (■ Disk 11)
■ Vol. 7, Issue 01 – ROM / Kernel Routines (■ Disk 12)
■ Vol. 7, Issue 02 – Games From The Inside Out (■ Disk 13)
■ Vol. 7, Issue 03 – Programming The Chips (■ Disk 14)
■ Vol. 7, Issue 04 – Gizmos and Gadgets (■ Disk 15)
■ Vol. 7, Issue 05 – Languages II (■ Disk 16)
■ Vol. 7, Issue 06 – Simulations and Modelling (■ Disk 17)
■ Vol. 8, Issue 01 – Mathematics (■ Disk 18)
■ Vol. 8, Issue 02 – Operating Systems (■ Disk 19)
■ Vol. 8, Issue 03 – Feature: Surge Protector (■ Disk 20)

Industry News

The following items are based on press releases recently received from the manufacturers. Please note that product descriptions are not the result of evaluation by The Transactor.

AmiEXPO in New York City

On October 10, 1987, AmiEXPO, a show dedicated exclusively to the Amiga and its aftermarket, will open to the public in New York City.

Among the exhibitors are Activision, subLogic, Central Coast, Brown-Wagh, Amazine Computing, AmiProject, Byte by Byte, Gold Disk, Ameristar, Manx, Lattice, Oxxi, and a host of others. Sounds like it'll be a good one!

Information on show hours and admission prices were not available at this time. For more information, contact AmiEXPO Headquarters, 211 East 43rd Street, Suite 301, New York, NY, 10017, 1-800 32 AMIGA. In NY call (212) 867-4663.

Special Amiga Software Offer

Commodore has announced a special software promotion for Commodore Computer Clubs and certified educators.

Commodore User Groups and their members, plus certified teachers and faculty members of Canadian schools, colleges and universities purchasing any Amiga computer between August 21 and October 31, 1987, will be offered a selection of brand-name software packages for a fraction of their usual retail price.

The "creativity software" package, with a suggested retail value of over $750 will be available for only $199, and includes:

- DELUXE PAINT II by Electronic Arts
- PAGESETTER, the desktop publishing system from Gold Disk
- AEGIS ANIMATOR by Aegis Development
- TEXTCRAFT PLUS word processor from Commodore-Amiga
- MARBLE MADNESS, the arcade game, also by Electronic Arts
- A 500XJ joystick from Epyx

The "productivity software" package, suggested retail over $1400, will be offered for only $399. This package contains:

- WORD PERFECT, a leading word processor from Word Perfect
- PAGESETTER DELUXE desktop publishing system with Laserscript, Fontset I, and Hyphenation modules, by Gold Disk (upgradable to Professional Page).
- SUPERBASE PERSONAL, an innovative database from Progressive Peripherals and Software
- MAXIPLAN 500, a multi-tasking spreadsheet program from Oxxi Inc.
- DIGA telecommunications program from Aegis
- DELUXE VIDEO presentation video and graphics system from Electronic Arts
- CLIMATE, an icon shell for Amiga DOS from Progressive Peripherals and Software

These combination Amiga and software offers will be available in more than 600 independent retail, and participating Canadian Tire and K-Mart stores across Canada. The promotion is aimed at influential users in our two most important markets – home/hobby and education.

For more information, please contact, Stan Pagonis or Katherine Dimopoulos, Commodore Business Machines, 3470 Pharmacy Avenue, Agincourt, Ontario, (416) 499-4292.

Benchmark Modula–2 from Oxxi, Inc.

Oxxi, Inc. has begun shipping Benchmark, a Modula–2 development system for the Amiga. The package includes an integrated compiler, linker and EMACS–based editor, along with separate version of the compiler and linker, Modula–2 standard libraries plus full Amiga support libraries, profiling, cross–reference and other utilities, extensive demos, and more than 700 pages of documentation.

Oxxi report average compilation speeds of 10,000 lines per minute with burst speeds of up to 30,000 lines per minute. Their proprietary linker is claimed to be similarly very fast compared with other high level language packages available for the Amiga. Benchmark Modula–2 is said by Oxxi to generate object code files of comparable size and execution speed to those created by the Aztec C compiler from Manx Software Systems Inc.

In addition to the basic $199 (US) package, three add-on products at $99 will be available at the time of Benchmark's official September release. These products will have extensive independent documentation and examples, but can only be used in conjunction with the basic package. They are:

- C Language Standard Library, including versions of printf, scanf, fopen, and many other standard C functions for easy porting of C programs to Modula-2.
- A special 'simplified' set of the Amiga standard libraries to allow quick, painless creation of Intuition windows, screens, gadgets and menus, plus convenient access to the IDCMP (Intuition's message port), the console device, speech synthesis, etc.
- IFF Libraries and a Graphic Image Resource Management library/utility. These will allow the programmer to process IFF files, and to include graphics data, in any of three formats, with program code for fast run-time access. The three formats currently supported are Intuition (BOB) format, Simple Sprites and Virtual Sprites.

For further information, contact Oxxi Inc., 1835-A Dawns Way, Fullerton, California 92631. Telephone (714) 999-6710.

DesignText, from DesignTech Business Systems Inc.

DesignText, a full-featured WYSIWYG (What You See Is What You Get) word processor for the Amiga, is scheduled to begin shipping October 1, 1987. The product makes use of the Amiga's standard Intuition user interface, with extensive use of gadgets, pulldown menus (most menu options also have keyboard shortcuts) and windows. Fast scrolling of the text can be accomplished either from the keyboard or with the mouse; horizontal scrolling is also supported beyond the standard 80 column screen width. Both interlace and non-interlace displays are available.

DesignText allows formatting in multiple columns, editing of multiple documents in multiple windows simultaneously, tabular input of figures, semi-automatic hyphenation and a selection of thirteen 8x8 fonts, each with a 12x24 equivalent for printer output. Printing can be done in up to four passes for additional resolution. IFF Graphics may be freely interspersed with text, and normal printing can be interspersed with the bit-mapped printing used for the special DesignText fonts.

Among the many other special features of DesignText are automatic index and footnote generation, widow and orphan control, special drivers for popular printers, allowing up to 292 dots per inch, the ability to import TextCraft, Scribble and Ascii files, two spell-checking modes (real-time and whole-document), complete on-line help, and optional terminal modules.

Apart from DesignText itself, the package also includes an easy-to-use data base called PeopleBase. PeopleBase is fully integrated with DesignText and can be called as a menu selection from within the word processor. It offers form letter and Little Black Book generation, address list and mailing label printing, client contact updating, multiple search criteria, and file security.

DesignText will sell for $79 (US) for advance orders, and at the regular price of $129 (US) upon release. For more information, contact DesignTech Business Systems Inc., #304-850 Burrard Street, Vancouver, British Columbia V6Z 2J1, Canada; or telephone (604) 669-1855.

JForth, from Delta Research Inc.

Delta Research has announced JForth for the Amiga, a software development environment based on the Forth '83 standard. The FIG and Forth-79 standards are also supported. Special toolboxes simplify development for the Amiga. Completed applications can be 'turnkeyed' and distributed without royalties.

JForth's code is claimed to be comparable in speed to C. Programs are compiled directly into machine code, unlike most Forths, which interpret tokens at run-time.

JForth is an interactive environment and a compiler. Any subroutine, variable, constant or data structure can be tested directly from the keyboard. By compiling incrementally, one can modify a program, compile, link and test in seconds. Utilities include a 68000 assembler and disassembler, search and sort routines, local variables, and floating point. Amiga structures and constants are predefined in ".j" files corresponding to the ".h" files used in C. Amiga library routines are called by name. An Object Oriented Development Environment (ODE) is also included. Example programs demonstrate graphics, HAM mode, speech synthesis, pull-down menus, etc. Source code for most of the system is included.

JForth is general purpose and can be used to develop games, business applications, music system, etc. JForth will run about four times faster on Amigas equipped with a 17 MHz 68020 processor. This makes it suitable for use in CPU-intensive engineering applications.

JForth is available at an introductory price of $99.95 US. Contact: Delta Research, 201 D Street, Suite 15, San Rafael, California 94901, (415) 485-6867.

LexCheck, from C.D.A Inc.

C.D.A. Inc. has announced LexCheck, a spelling checker for the Amiga. LexCheck works with Textcraft, Scribble! and Notepad formatted files as well as all ASCII text files. The master dictionary contains 100,000 words and you may add your own words to the auxiliary dictionaries. Because the dictionary always resides on disk, LexCheck uses less than 100K of RAM, and thus can be run simultaneously with most word processors.

In addition to standard English vocabulary, LexCheck also recognizes a wide variety of proper names, place names and technical terms. C.D.A. reports that a two-page document can be checked in under one minute.

LexCheck runs under Workbench 1.2, and retails for $42.95 (US). Contact C.D.A. Inc., P.O. Box 1052, Yreka, California 96097, telephone (916) 842-3431.

VideoScape 3D, from Aegis Development Inc.

Aegis has released VideoScape 3D, a set of three programs for the Amiga that permit the creation of 3-D television quality graphics and animations. The package requires a minimum of 512K RAM, with more preferred.

The three programs that make up the package include Designer 3D, for making 3D objects; PlayANIM, for playing back animations in real time; and VideoScape 3D, for making the finished video, plus utilities for creating common geometric shapes such as spheres, cones, rectangles and fractal landscapes.

3D objects are created either by numeric entry of X, Y, Z coordinates, or by one of the supplied utilities, or by using Designer 3D's three-windowed point-and-click method. With D3D, there are three windows representing the front, side, and top view of your object as you create it. Scaling, numeric value display, and numeric entry help maintain accuracy as the object is drawn. When finished, a motion file can be loaded and the object is passed to the preview window. Here, in real time, the object can be shown in motion from all sides. Each frame of the motion file is recorded in RAM and played back at an adjustable speed.

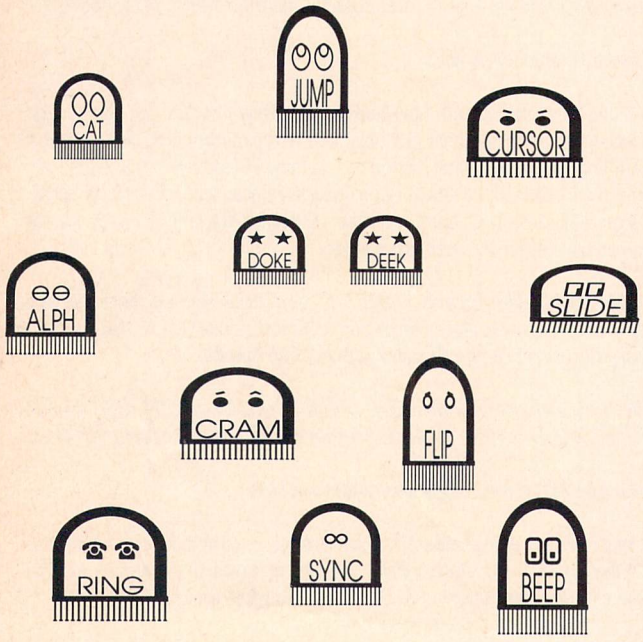
Once the objects are created, they are loaded into VideoScape's main control panel. Here the scenes are put together. Details regarding camera and object motion, backgrounds, foregrounds, horizon and other information are determined, and a complete scene is created. A VCR can be hooked up to record the scene one frame at a time, or a few seconds at a time using the PlayANIM module.

To create an ANIM file requires at least one megabyte of RAM; however, many ANIMs will play back on a 512K Amiga. An ANIM file can compress a 40K frame into less than 1K. The ANIM compressed file format is available at no charge to interested parties.

VideoScape 3D, $199.95 (US) from Aegis Development Inc., 2210 Wilshire #277, Santa Monica, California, telephone (213) 392-9972.

New! Improved! TRANSBASIC 2!

with SYMASS™



"I used to be so ashamed of my dull, messy code, but no matter what I tried I just couldn't get rid of those stubborn spaghetti stains!" writes Mrs. Jenny R. of Richmond Hill, Ontario. "Then the Transactor people asked me to try new TransBASIC 2, with Symass®. They explained how TransBASIC 2, with its scores of tiny 'tokens', would get my code looking clean, fast!

"I was sceptical, but I figured there was no harm in giving it a try. Well, all it took was one load and I was convinced! TransBASIC 2 went to work and got my code looking clean as new in seconds! Now I'm telling all my friends to try TransBASIC 2 in *their* machines!"

• • • • • • •

TransBASIC 2, with Symass, the symbolic assembler. Package contains all 12 sets of TransBASIC modules from the magazine, plus full documentation. Make your BASIC programs run faster and better with over 140 added statement and function keywords.

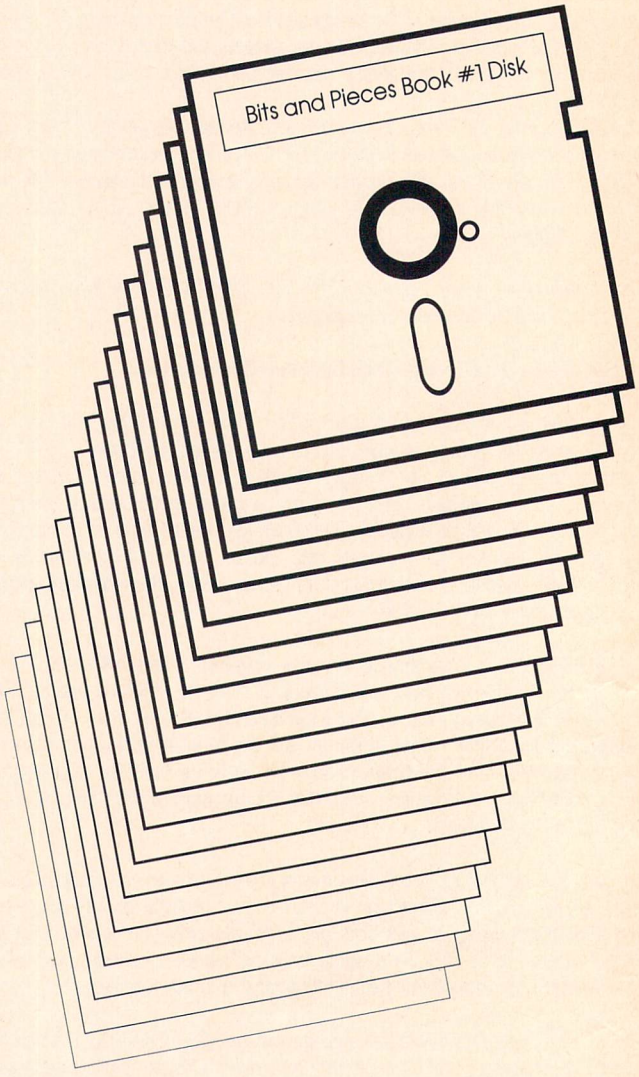
Disk and Manual $17.95 US, $19.95 Cdn.

(see order card at center and News BRK for more info)

TransBASIC 2

"Cleaner code, load after load!"

Bits & Pieces I: The Disk



From the famous book of the same name, Transactor Productions now brings you *Bits & Pieces I: The Disk!* You'll **thrill** to the special effects of the screen dazzlers! You'll **laugh** at the hours of typing time you'll save! You'll be **inspired** as you boldly go where no bits have gone before!

"Extraordinarily faithful to the plot of the book. . . The BAM alone is worth the price of admission!"
Vincent Canbyte

"Absolutely magnetic!!"
Gene Syscall

"If you mount only one bits disk in 1987, make it this one! The fully cross-referenced index is unforgettable!
Recs Read, New York TI$

WARNING: Some sectors contain null bytes. Rated GCR

BITS & PIECES I: THE DISK, A Mylar Film, in association with Transactor Productions. Playing at a drive near you!

Disk $8.95 US, $9.95 Cdn. Book $14.95 US, $17.95 Cdn.
Book & Disk Combo Just $19.95 US, $24.95 Cdn!

THE TIME SAVER

Type in a lot of Transactor programs?
Does the above time and appearance of the sky look familiar?
With The Transactor Disk, any program is just a LOAD away!

Only $8.95 US, $9.95 Cdn. Per Issue
6 Disk Subscription (one year)
Just $45.00 US, $55.00 Cdn.
(see order form at center fold)

Now Amiga Owners Can Save Time Too!

Transactor Amiga Disk #1, $12.95 US, $14.95 Cdn.

All the Amiga programs from the magazine, with complete documentation on disk, plus our pick of the public domain!

THE WORLD OF COMMODORE

The 1986 Canadian World Of Commodore show was the largest and best attended show in Commodore International's history. With 350 booths and attendance of over 38,000 users it was larger than any other Commodore show in the World — and this year's show will be even larger.

World of Commodore is designed specifically to appeal to the interests and needs of present and potential Commodore owners — from hardware to software, Business to Personal to Educational.

Exhibitors: Write or phone today to find out how you can take part in the World's largest Commodore Show.

For information contact: The Hunter Group Inc. (416) 595-5906

INTERNATIONAL CENTRE

DECEMBER 3-6, '87

TORONTO