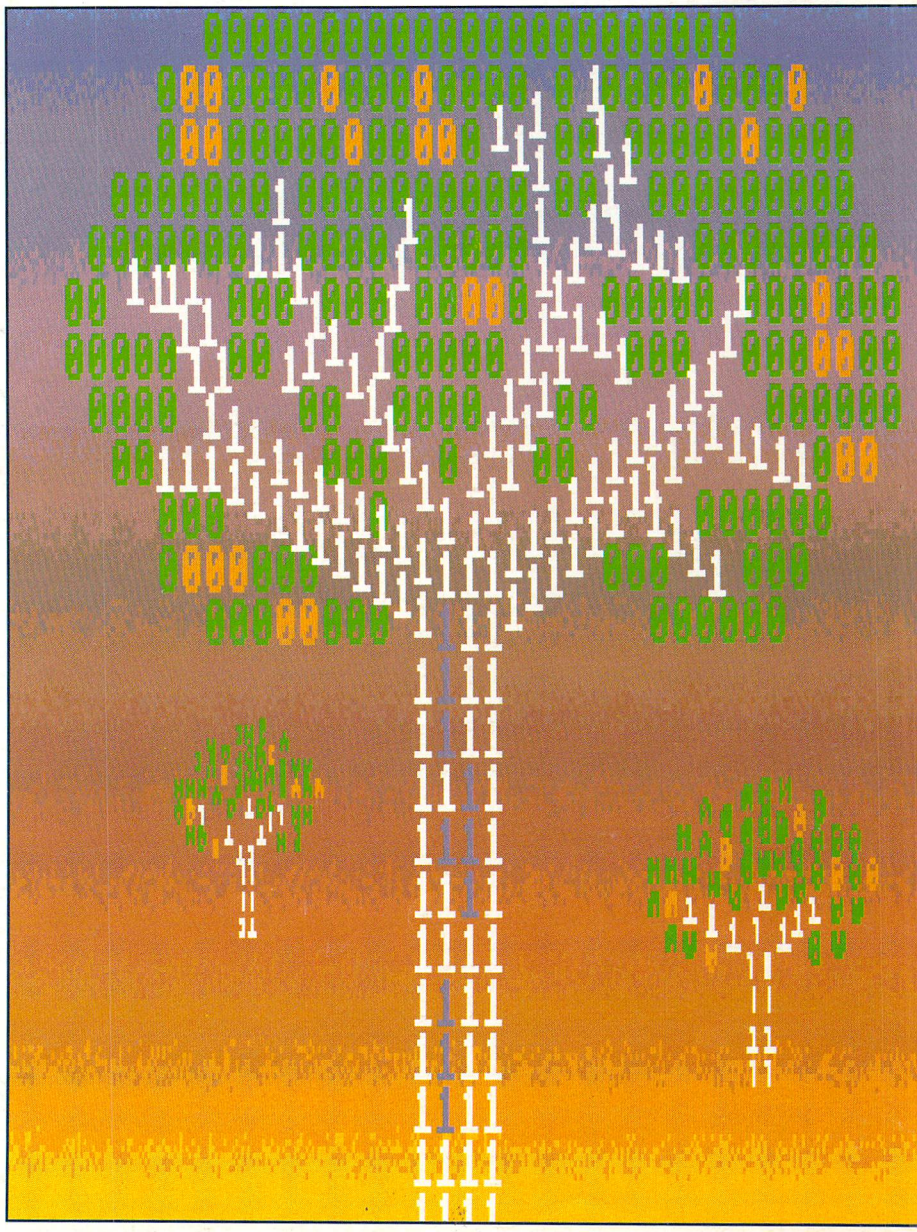
TRANSACTOR: THE MAGAZINE FOR COMMODORE PROGRAMMERS MARCH 1988: VOLUME 6, ISSUE 5

Transactor

Canada $4.25
USA $3.50



Fast String Searches with Binary Trees

0 29074 12121 03

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!!

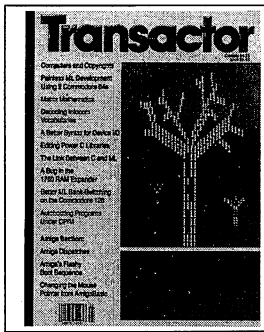
See Order Card at Center

Volume 8
Issue 5

Transactor

Amiga Section

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



ABOUT THE COVER: The tree was drawn using Deluxe Paint II, then simply stored on disk, packed in an envelope, and shipped to the ImageSet Corporation in San Francisco, CA. It seems the Amiga will do just about anything, and colour separations on a micro is an application that's about as cost and time effective as you can get. Using a program called "ColorSet", the IFF file containing our picture can be resized vertically or horizontally before the separations are generated.

Transactor
The Magazine for Commodore Programmers

Editor-in-Chief
Karl J. H. Hildon

Publisher
Richard Evers

Technical Editor
Chris Zamara

Submissions Editor
Nick Sullivan

Editorial Assistant
Moya Drummond

Customer Service
Jennifer Reddy

Contributing Writers

Ian Adam
Steve Ahlstrom
David Archibald
Jack Bedard
Paul Blair
Neal Bridges
Bill Brier
Anthony Bryant
Jim Butterfield
Dale A. Castello
Betty Clay
Tom K. Collopy
Don Currie
Robert V. Davis
Elizabeth Deal
Frank E. DiGioia
Chris Dunn
Michael J. Erskine
Jack Farrah
Mark Farris
Jim Frost
Miklos Garamszeghy
Eric Germain
David Godshall
Michael T. Graham
Eric Giguere
Thomas Gurley
Tim Grantham
Patrick Hawley
Adam Herst
Thomas Henry
John Houghton
Robert Huehn
David Jankowski
Clifton Karnes
Lorne Klassen
Jesse Knight
Gregory Knox
David Lathrop
James A. Lisowski
Richard Lucas
Scott Maclean
Steve McCrystal
Chris Miller
Keath Milligan
Terry Montgomery
Ralph Morrill
D.J. Morriss
Michael Mossman
Bryce Nesbitt
Gerald Neufeld
Noel Nyman
Matthew Palcic
Richard Perrit
Larry Phillips
Steve Punter
Raymond Quirling
Doug Resenbeck
Tony Romer
Herb Rose
Dan Schein
E.J. Schmahl
David Shiloh
Darren J. Spruyt
Aubrey Stanley
David Stidolph
Richard Stringer
Anton Treuenfels
Audrys Vilkas
Jack Weaver
Geoffrey Welch
Evan Williams

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	– e
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	– i	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

Mondays, Wednesdays and Fridays ONLY

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:

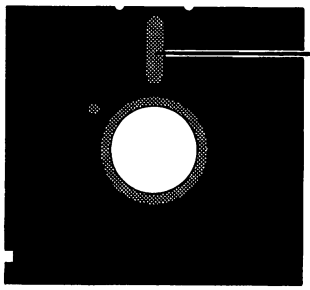
In Canada:
Ingram Software Ltd.
141 Adesso Drive
Concord, Ontario
L4K 2W7
(416) 738-1700

In the U.S.A.:
IPD (International Periodical Distributors)
11760-B Sorrento Valley Road
San Diego, California
92121 (619) 481-5928
Ask for Dave Bruescher

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, 04, 05

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, 8250, 1571, or 1581 diskettes with WordPro, PaperClip, Pocket Writer, WordCraft, Superscript, (actually, just about any word processor files) or SEQ text files, or Amiga format 3½ diskettes with ASCII text files. Program listings including BITS subsmissions 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

Socially Unacceptable

Have a look at the first item under "Industry News" in the New BRK section. It's a report of a federal court case underway in New York state. Should judgement be found for the plaintiff, the defendant will be convicted of operating a pirate bulletin board system. More significant than that is the fact that his or her parents will be held responsible.

I, for one, am rooting for the plaintiff. And if they should win? Send them on a walk down the gangplank if that's suitable (and I'm in a *good* mood right now). But since that doesn't happen in our "civilized age", the court will need to think of something else. Punishment of equal significance I would hope. Punishment that will make others think twice about supplying the public at large with commercial software. And not just over the phone lines. . . hand-to-hand transfers too. We need a deterrent. A deterrent commensurate with the crime. One that deters the act because getting caught would be socially unacceptable. Those involved with this case have the opportunity to set a valuable precedent here. . . one that may prove to be the best protection yet against software piracy.

Looking at software stealing over the years, it seems little else has proven very effective for very long. Early packages used a ROM installed in a spare socket, but it wasn't long before the program stopped looking at the ROM. Same with the dongle. Then there was disk surface protection. Thick manuals. They all had one thing in common, though. Making a copy was something to be proud of. . . friends admired you, even if they weren't the recipient of the duplicate. The skill required was quite captivating. Like drinking and driving, that must change.

Unfortunately, changes like that require money. . . a lot of money. Changing the public's attitude toward anything, at least on this continent, *is* possible, but it usually means relentless, expensive campaigns. Even then, there will always be those who choose to ignore the consequences. And since nobody is suffering physically when a program is passed along, a public campaign is unlikely. It looks like we'll have to settle for the embarrassment of a conviction that can be used publicly to equate software pirates with the common thief.

But convictions are expensive. The revenue lost by one outfit to even a hundred downloads probably doesn't compare to the attorney's fee. Add up the total losses among all the manufacturers whose titles are available for capturing though, and the lost revenue becomes potentially staggering. The most popular pirate boards are those with the largest selection. Naturally the plaintiff cannot collect on behalf of all the others, so it's up to the courts to assess the damage and determine suitable punishment.

The software industry is easily in the billion dollar range per year. That money goes a lot farther than just over the counter and into the till. Every copy of any commercial program that isn't sold means the author loses, the manufacturer (our advertiser) loses, the retailer (who also sells our mag) loses, and the consumer pays higher prices. When these outfits can't sustain themselves any longer, they go under. The employees lose, and the customers lose when they can no longer get support for the products they bought. Who else? How about the essential services that every company needs? It's time the authorities took notice. A rumour I heard recently suggests they are. Perhaps by next issue I'll have more on that.

As far as this latest round goes, word is that the Doe family stands a good chance of losing their telephone service. It's within the court's power to order it and the deterrent value would have far more impact than even the most reasonable fine. I would hope the telephone company won't have a problem with that either. We pay lots of dineros to the phone people and it's my guess that many of the software companies that have gone under were paying more than us.

It's also my contention that the phone company could do more to eliminate the pirate BBS. I don't expect them to go around snipping phone lines. Simply alerting the proper authorities would be enough on their part. It would be interesting employment for someone who not only enjoys BBSing but also wants to preserve it, and it could certainly be considered as revenue protection for the phone company. But many of you will disagree, I know. Because if Ma Bell gets involved, it's very possible that the BBS as we know it will become extinct. BBS operators who don't even take donations will suddenly be paying to offer their service, which might have been completely on the level and no more than a hobby. There isn't room here to describe how the phone people could do this job right, but I think it's worth looking at.

The outcome of Weaver vs. Doe will be extremely important. I recently saw a BBS directory that offered the entire product line of one firm. The company is aware of the BBS and in the process of reacting. I find it most distasteful that anyone could participate in something so evil, especially when they must know how much it can hurt an honest manufacturer trying to make an honest buck. And based on that, I do hope more pirate boards are caught, eliminated, and penalized. And perhaps one day enough Doe's will have lost their telephone lines to make policing the BBS operators unnecessary.

It's not the solution, but it's a start. Hand-to-hand transfers are still the biggest problem. But if this angle can be successfully labeled "socially unacceptable", it's a step closer to changing the attitude towards piracy at all levels. I plan to do my part. You can do yours by boycotting the pirate BBS. Also, I'm sure manufacturers would be most interested in the whereabouts of boards offering their software. And *I* intend to make it possible for more firms to react.

I should have more on the outcome of Weaver vs. Doe by next issue. Let's just hope it doesn't set a precedent that renders us powerless to do anything but watch our software industry die. . . and that includes the Amiga.

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 3072,1 to enable the C128 version (off: SYS 3072,0)

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

Note: If a report code is missing (or "--") it means we've edited that line at the last minute 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 VERI-

FIZER 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.

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
```

Introducing The Standard Transactor Program Generator

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

Once the program is typed in (check the verifizer codes as usual when entering it), save it on a disk for future use. Whenever you type in a program generator (for example listings 5 and 6 from the article "Interfacing two Commodore 64s" in this issue), the listing will refer to the standard generator. Load the standard generator you've saved, then type the lines from the listing as shown. The resulting program will include the generator code and be ready to run.

When you run this new generator, it will create a machine language program on disk that can be loaded (load"filename",8,1) and executed with a SYS command. The machine language program is described in the related article, and the generator is just an easy way for you to create it using the standard BASIC editor at your disposal. After the machine language file has been created, the generator is no longer needed. The standard generator, however, should be kept handy for all future Transactor type-in program generators.

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

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

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

Figure This One Out!

Here's a challenge for all who think they know the 64's innards pretty well: examine the following program:

```
1 print"*";: poke NUM,0
```

The trick is to fill in "NUM" with a value that will cause the program to fill the entire screen with asterisks. Those who figure out the POKE win nothing but the satisfaction of having solved the puzzle. The first one who can figure out TWO solutions, however, gets a Transactor bits book as a prize. Just to save you from trying too hard and possibly losing your sanity, it's only fair to tell you that we know of no second solution.

Sprite Memory Display

Richard Lucas
Los Angeles, CA

This bit is not just a dazzler, it's also educational. The VIC-II chip in the Commodore 64 can use any section of memory to output a display, including page zero. This program combines this feature with sprites to create a dynamic bit-mapped display of the first two pages of memory. When run, it sets up eight sprites, then loops so that you can see zero page activity during the execution of a BASIC program. Press any key to exit the proram. The sprites will remain on the screen. Notice how much less activity there is in immediate mode? See how the display changes as you move the cursor, evaluate an expression, list a program, etc. When you're finished, RUN 999 To restore your computer to normal.

```
MP  10 rem zero page display  rl 5/10/84
NE  60 poke 53280,0:poke 53281,0:print"[md. grey]"
MO  100 vi=53248
OE  110 poke vi+21,255
OF  120 poke vi+23,255
KH  130 poke vi+29,255
PK  140 poke vi,30: poke vi+1,65
HK  150 poke vi+2,90: poke vi+3,65
ON  160 poke vi+4,150: poke vi+5,65
EP  170 poke vi+6,210: poke vi+7,65
BG  180 poke vi+8,30: poke vi+9,135
IM  190 poke vi+10,90: poke vi+11,135
DM  200 poke vi+12,150: poke vi+13,135
AN  210 poke vi+14,210: poke vi+15,135
DD  220 for i=0 to 7: poke 2040+i,i: next
DP  230 for i=0 to 7: poke vi+39+i,i+7: next
BN  250 print"[clr]page zero"
FC  260 print"[8 crsr down]page one[6 crsr down]"
OG  270 get a$: if a$="then 270
OA  280 print "[2 crsr down]goto 999[2 crsr up]"
CC  290 end
IP  999 poke 53248+21,0: print "[clr]";
```

File Track and Sector

Paul Blair
Canberra, Australia

This routine came out of desperation, when I wanted to find (quickly) where a certain program was stored on disk. I tried reading the directory, but that all takes time. Too slow.

Then I figured that the Disk Operating System does it all anyway when it opens a file. Hmmmmm. The trick is to find which internal buffer has been set when the OPEN command is issued. DOS uses location $F9 in the disk drive to store the number of the buffer (0–5 in the 15** drives) that will be used. Line 20 reads the buffer number. Knowing this, we have identified which of the "pairs" of track and sector values stored between $06 and $11 in the drive we want for the file in question. Buffer 0 uses $06 and $07, buffer 1 uses $08 and $09, and so on. A quick fix in line 30, and we can get the values we want.

```
IE  5 rem find track and sector of a file
FE  10 z$=chr$(0): input"which file";nm$: open 15,8,15
PM  20 open 2,8,2,nm$: print#15,"m-r"chr$(249)
        chr$(0)chr$(1)
MG  30 get#15,a$: bf=6+2*asc(a$+z$)
LO  40 print#15,"m-r";chr$(bf)chr$(0)chr$(2)
CF  50 get#15,a$,b$: tr=asc(a$+z$): sc=asc(b$+z$)
GF  60 close 2: close 15: print"track";tr;"sector";sc
```

File Load Address Changer

Darryl Brimner
Windsor,Ont.

In many instances it is desirable to have versions of relocatable data and programs that have different starting addresses. In other

situations it may be convenient to have the ability to change the starting addresses on disk. For instance Ultrafont stores its character sets at $7000, while Paperclip uses $0800. The following program gives the user the ability to change the load address of a program file (PRG) on disk. Note the way the track and sector of the file is obtained. *(Not the same way Paul Blair does it in the previous bit – CZ)*

```
BJ 1000 rem change load address of file
ON 1010 input"filename>";fs$
FN 1020 print:input "new load address>";na
IE 1030 open 2,8,0,fs$:close2:open 15,8,15
       :print#15,"m-r";chr$(144);chr$(2)
CH 1040 get#15,s$:print#15,"m-r";chr$(147);chr$(2);
       chr$(2):get#15,t$,b$
IL 1050 open 2,8,2,"#":print#15,"u1 2 0";asc(t$);asc(s$)
KJ 1060 print#15,"b-p 2";asc(b$):get#2,f$,t$,s$
PL 1070 print#15,"u1 2 0";asc(t$);asc(s$)
       :print#15,"b-p 2 2"
HK 1080 hi=int(na/256):lo=na-hi*256
JP 1090 print#2,chr$(lo);chr$(hi);
       :print#15,"u2  2  0";asc(t$);asc(s$)
LK 1100 close2: close15
```

File Stripper — **Eric Giguere, Waterloo, Ont.**

When submitting articles to the Transactor, the editors prefer to have a copy of the article on disk as well as on paper. But any word processor files you use are bound to have embedded commands in them, commands which are useless for the Transactor's editorial purposes. It's best to send a simple file without any of these commands. So to make things simpler, I wrote a small program that would take a Commodore 64 EasyScript file and automatically strip out the embedded commands. The program should be easily modifiable to suit your own wordprocessor. The lines specific to EasyScript are lines 120 to 160. Line 120 is special because it intercepts EasyScript command lines and sets a no-print flag (the flag is reset when the end-of-line is encountered). Apart from this, the program simply ignores characters that aren't valid alphanumerics or punctuators.

```
PG 100 rem easyscript file stripper
AE 110 rem by eric giguere
GJ 120 rem
BE 130 print "[CLR DWN] input filename: ";: gosub 270
      : fi$=left$(in$,16)
BL 140 print "[DWN] output filename: ";: gosub 270
      : fo$=left$(in$,16)
BM 150 fl=0: print "[DWN] opening the files. . ."
GB 160 open 15,8,15,"i0": gosub 280: if e then 300
AH 170 open 2,8,2,"0:"+fi$+",s,r": gosub 280
      : if e then 300
MJ 180 open 3,8,3,"0:"+fo$+",s,w": gosub 280
      : if e then 300
AJ 190 get#2,a$: a=asc(a$+chr$(0)): ok=st
LO 200 if a=128 then get#2,a$: if a$="*" then fl=1
      : goto 250
CP 210 if fl=1 and a=13 then fl=0: goto 250
CO 220 if (a<32 and a<>13) or a>218 then 250
JC 230 if a>95 and a<192 then 250
II 240 if fl=0 then print#3,a$;
NI 250 if ok=0 then 190
PD 260 close 2: close 3: close 15: end
HG 270 open 1,0: input#1,in$: close1: print: return
OJ 280 input#15,e,e$: if e<20 then e=0
OD 290 return
CD 300 print "[DWN] disk error: ";e;e$: goto 240
```

No-Question Mark Input — **Keath Milligan, Austin, TX**

Input from BASIC without the question mark prompt can be accomplished by fooling BASIC into thinking it is receiving input from something other than the keyboard. When the input routine is called, it checks location 19; if this location contains a zero, the routine will know the keyboard is being used and will do a few extra things, such as printing the question mark.

Try this:

```
10 poke 19,1: input"prompt";a$: poke 19,0
```

The system uses the same input routine in immediate mode as the one used by the INPUT statement, so the POKE 19,0 is needed to put things back to normal.

Binary Sorting "On the Fly" — **Don Ellis, Winnipeg, Manitoba**

Further to 'Sorting on the Fly' in the September 87 Bits and Pieces:

Sorting data as it is entered can also be accomplished with a binary-sort algorithm instead of a sequential substitution as used in the illustration given. Although the sequential algorithm in Martin Hofheinz's program (lines 80–90) is indeed faster than a binary sort (or at least, than my binary sort) for small lists, in my tests the binary sort takes over on speed somewhere in between sixty and seventy total – I mystically suppose the exact point would turn out to be 63. The relative speed difference curves rapidly apart from there, so that with a 200-member array, the sequential sort takes 30% longer than the binary.

Even more important, the sequential sort is faster in a region where they are both virtually instantaneous; shortly past the threshold of noticeability of processing time – which I take to be about half a second – the binary sort is faster anyway. Finally, the binary sort has a much narrower range of variation. This means, for example, that the worst-case situation in a 200- member array sort can take *two or three times as long* with a sequential algorithm – a noticeable difference indeed. To examine the difference, substitute the following lines 80–90 for those in the original program; then compare the time required to sort alphabetically (the new binary sort) with that required to sort by age (the sequential substitution). The effects will be most noticeable with arrays over 100.

```
KF 80 h=(j-1)/2: g=0
CI 82 if (t$(j)<t$(a(h))) then h=int((h+g)/2): goto 86
IF 84 g=h: h=int((h+j)/2)
BG 86 if h-g>0 then 82
MP 88 h=h-(t$(j)>t$(a(h))): g=j-1
LM 90 if g>=h then a(g+1)=a(g): g=g-1: goto 90
```

Handy Hexer — **Brian Mcintyre, Vancouver, BC**

One of the neatest and easiest subroutines I have ever used for printing hex numbers is the following.

```
IH  10 b=int(a/256):gosub 30
KK  20 b=a-b*256:gosub 40:return
ON  30 print "$";
NI  40 x=int(b/16):gosub 50:x=b-x*16
AL  50 printmid$("0123456789abcdef",x+1,1);
IF  60 return
```

The routine may be entered at four places, line 10 for two byte numbers (0 to 65535), lines 30 and 40 for one byte numbers (0 to 255), and line 50 for single digits (0 to 15). These examples show the different ways to use it:

Input	Output
a=49152: gosub10	$C000
b=234: gosub30	$EA
b=234: gosub40	EA
x=10: gosub50	A

There is a good deal of choice as to how to use this routine, and any lines above the line called are unnecessary. Note that this routine leaves the next print position immediately after the number that was printed.

Microtrace — **Peter Lottrup, Buenos Aires, Argentina**

Here is a short trace utility which will display the current BASIC line number in reverse field on the top left corner of the screen. It is placed in the cassette buffer, and activated with SYS 828. It can be disabled with SYS 831.

The program below is a modified version of Mr. Lottrup's original microtrace, *with an added feature: you can slow down your BASIC program while it is being traced by holding the CTRL key down. When CTRL is released, the program continues at full speed. This makes* microtrace *even more useful for debugging tricky programs. – CZ*

```
DO  10 print" microtrace
NB  20 print"sys 828 to trace program lines
LC  30 print"hold ctrl key to slow trace down
BE  40 print"sys 831 to turn microtrace off
GK  50 :
BO  60 for i=828 to 934: read a
EL  70 poke i,a: ch=ch+a: next i
IL  80 if ch<>11716 then print"data error!"
KF  90 end
LF  1000 data 76, 66,  3, 76, 77,  3, 169,  88
CB  1010 data 141,  8,  3, 169,  3, 141,  9,  3
JJ  1020 data 96, 169, 228, 141,  8,  3, 169, 167
FO  1030 data 141,  9,  3, 96, 166, 58, 134,  99
NM  1040 data 232, 240, 59, 165, 57, 133, 98, 162
AD  1050 data  0, 160,  8, 169, 176, 157,  0,  4
MH  1060 data 56, 165, 98, 249, 157,  3, 72, 165
PJ  1070 data 99, 249, 158,  3, 144, 10, 133,  99
NG  1080 data 104, 133, 98, 254,  0,  4, 176, 232
PC  1090 data 104, 232, 136, 136, 16, 221, 173, 141
MF  1100 data  2, 240, 11, 162, 32, 165, 162, 197
BH  1110 data 162, 240, 252, 202, 208, 247, 76, 228
HB  1120 data 167,  1,  0, 10,  0, 100,  0, 232
PN  1130 data  3, 16, 39
```

C128 BITS

Monitor Dump to Sequential File — **Philip Herold, Seattle, WA**

When examining ROM or a long program with the 128's monitor, it's hard to keep track of where you are, especially when you're on level six of a series of nested subroutines. One solution is to dump the whole thing to a printer. Better, in my opinion, is to create a sequential file of the monitor dump, load it into a word processor, take advantage of its search capabilities and comment the file to your heart's content.

Here's how to create such a sequential file, dumping the first two pages of BASIC ROM as an example. Type in direct mode:

```
open1,8,2,"filename,s,w": cmd8
monitor
d f4000 f4200
```

When the cursor reappears, type

```
x
print#8: close8
```

. . .and that's all there is to it. The same technique works on the 64, using a monitor program (accessed with the appropriate SYS command).

50 line, 80 column display on the C128 — **Darrel Grainger, Toronto, Ont.**

The C128 80 column screen is capable of displaying 50 rows of text. The BASIC screen editor routines will only use 25 rows, but you can take advantage of the 50 line display in your own programs.

This program will alter the 8563 VDC chip to display 50 rows. You won't be able to use all 50 lines when programming, but this gives you an idea of what the display looks like and shows you what values to use for your programs.

```
    0 rem sys write,val,reg
    1 :
    2 rem write = routine to write to 80 column registers
    3 rem val   = value being written to register
    4 rem reg   = reg to which the value is being written
    5 :
CA  100 sys 52684,128,0  :rem 126 horizontal total
HM  110 sys 52684,104,2  :rem 102 horizontal sync position
FJ  120 sys 52684,137,3  :rem 73  bit 7-4 vertical sync width
IO  130 sys 52684,64 ,4  :rem 32  vertical total
MP  140 sys 52684,50 ,6  :rem 25  vertical displayed
```

```
JP | 150 sys 52684,57 ,7  :rem 29  vertical sync position
GM | 160 sys 52684,3  ,8  :rem 0   interlace mode
GI | 170 sys 52684,7  ,25 :rem 71  bit 6 disable attributes
FA | 180 sys 52684,16 ,26 :rem 240 back/foreground colors
```

(The value immediately after the REM statement on each line is the original value for each register.)

Line 100 – This modifies the horizontal total to make the display more readable.

Line 110 and line 150 – When you alter the display, the screen will move around. These registers will help to centre things on the screen (you could use the horizontal/vertical positioning on the monitor but that would mess up your 40 column picture).

Line 120 – This helps clear up flicker. The high nybble is doubled as the Programmer's Reference Guide suggests.

Line 130 and 140 – This sets the number of displayed lines to twice as many as usual. Line 130 is the number of lines on the screen including the space in the top and bottom borders. Line 140 is just the lines displayed.

Line 160 – This turns on interlace mode, necessary for a 50 line display.

Line 170 – I disabled attributes because the bottom 25 lines, which I could not access from the BASIC screen editor, were flashing annoyingly. This has the side effect, among other things, of not allowing upper/lowercase characters. If you wish to use the upper/lowercase character set, delete lines 170 and 180 and set the character colour in the usual way.

Line 180 – Because there are no attributes, all screen colour defaults to the value in register 26. The value 16 used here produces grey text on a black background.

This is the best 50 line display I could produce. If someone comes up with a better set of values to reduce flicker, I would appreciate hearing about it.

Windows on the 128 — **Ian Adam, Vancouver, BC**

If you've done much programming on the 128, you know how useful the windows can be in setting up different areas of the screen for different purposes. You also know that windows can be created by either ESCape sequences or the WINDOW command, and are cancelled by printing HOME twice.

Well, here's a twist. If you open two windows in succession without PRINTing in the first, *both* windows are cancelled and you get the full screen! Something like this:

```
10 window 20,5,35,8
20 rem. . . in a loop waiting for an event
30 window 10,10,35,15
40 print"cursor should be at row 10 column 10"
```

Whenever you open a window, the cursor is automatically placed in the home position. It seems that, when you open a second, the screen editor treats this as two successive HOMEs, and dutifully cancels your window. This could happen inadvertently in a program, with unfortunate results.

Let's say you use several windows on the screen – one for a title, one for input, one for status, etc. You activate the input window and leave the cursor there to await a message from the user. Before that is received, however, some other event changes the status. Your program opens the status window – or so it thinks! Instead, all windows are cancelled and your status message gets dumped all over your nicely centred title.

How to avoid the problem? Number one: always print something when you open a window, even if it's just a "cursor up". Number two: put a ",1" after the WINDOW command, to clear the window. Number three: If you can manage it, print the less-convenient escape codes to define your window. It's not hard to deal with the problem, once you know it exists.

AMIGA BITS

The Other Drag Bar

When you've sized a window so narrow that there's no drag bar left to grab it by, there's still a way to drag the window without resizing it again! Look to the right of the window–to–front gadget; a few pixels of title bar overhang beyond the right of the gadget! This is actually a tiny strip of drag–bar, two pixels wide. Point right at the line that forms the right side of the front gadget, and you can grab the window and drag it around from there.

Awrite File Transfers — **Glenn Wiorek, Chicago, IL**

There is an undocumented switch in Awrite, the program that transfers files from the MS–DOS side of a bridge–card equipped Amiga 2000 to the Amiga side. Awrite is found on the GW Basic disk that comes with the A2088 Bridgeboard. Awrite has problems transferring executable files, transferring only the first 256 to 512 bytes.

In order to transfer any file properly, you have to use the undocumented /B switch to put Awrite into binary transfer mode. The format for this command (in an MS–DOS window) is:

```
AWRITE [drive:][path]filename drive:[path]filename /B
```

Example: `AWRITE a:\bin\test.arc DF0:test.arc /B`

The above command would transfer the file "test.arc" from the MS–DOS A: drive in the "bin" directory to the root directory of Amiga floppy drive 0.

Amiga 500 Serial and Parallel Ports — **Richard Lucas, Los Angeles, CA**

Pages A–4 and A–5 of the "Introduction to the Commodore Amiga 500" manual show pictures of the Amiga serial and parallel ports, respectively. The manual doesn't say this, but the pictures show the cable connectors, not the ports on the Amiga 500. In addition, there is a warning on page A–3 that implies that you need a special cable to connect the Amiga 500 to your printer through the parallel port. *(This is true for the Amiga 1000 – CZ)* Actually, I have found that my standard IBM parallel printer cable (DB25 male to Centronics 36 male, with all DB–25 pins present except 20 through 25) will work just fine with printers that have standard Centronics inputs.

Letters

A question on copyright: First I'd like to thank you for publishing my reply letter to the Jordan Rolltop Stand (Transactor, Volume 6, Issue 6). You wished me success then so I'm reporting that I did get three requests for further information and one subsequent order making my year ending December 31, 1986 register a net profit of $15.90. Not really enough to allow me to quit my job, but the experience was invaluable: when I do launch my business, I will be better prepared to handle the type of requests likely to come in. (If I can comprehend the requests of hackers who aren't really into drills, saws, etc., then I should fare well with the cabinetmakers of the world.) Thanks again.

Second is a question concerning copyright. (Oh no, not again!) I recently did some extensive reworking on Commodore's "EasyMail 64" for a local businessman, speeding disk access, adding features, and even cutting program size by about 40 per cent. My question is this: can I legally offer this as an 'upgrade' product or would I get nailed for not "reading the program into memory of a computer solely for the purpose of executing the program. . ."? I was thinking of a deal where a legal owner of EasyMail could send me the disk and I would copy the new program to it for a couple of dollars, but I try to "toe the line" in regard to copyright and don't need any trouble either. As a sidelight on this, one of the things I did to EasyMail was to use an ML sort routine for the listing routines. Unfortunately, the routine I used came from an old issue of Compute! and, being the type they are, I have to rewrite it myself so I don't violate their copyright. Any comments on this would be appreciated.

Third is a comment on the circulation situation that received attention in the letters column for a while. Don't get too upset when people complain about your modest circulation. The T. will never be as widely circulated as Compute!, their Gazette, or the others until you quit giving readers material to chew on (and digest and grow up to be big, strong programmers) and start spoon-feeding us ready-made programs in a format that teaches (and requires) nothing but some time spent typing them in. Every month I get more disappointed with Compute! Publications and long for more of the T. Keep up the good work.

Matthew R. Strange, Mansfield, Pennsylvania

Glad we could help you establish such a lucrative business, Matthew. Next time you're passing by in the Rolls, why not stop in at the office and treat us to a few of those fat Cuban cigars you're washing your martinis down with nowadays?

We can't see that you have a copyright problem with Easymail – what people do with their bought-and-paid-for commercial programs is their own affair; as far as we know it is perfectly legal to sell an aftermarket upgrade. Several upgrade packages for Speedscript are commercially available, with no argument from COMPUTE! who, as you note in your letter, are not generally shy about proclaiming the sanctity of their copyrights.

As for circulation – you're right, ours will probably never go into orbit. We'd still like to reach the stratosphere, though, and we're hoping to see a lot of improvement in that area now what we're getting back into retail distribution.

Transformer Modification for MC68010 Upgraded Amiga: Back in the September 1987 issue of Transactor (Volume 8, Issue 2) my article called "Upgrading the Amiga 1000 to 32 bits" was published. It told how to replace your MC68000 with an MC68010 and also how to modify your disks so that 99% of your programs would run with the MC68010.

Then, in the November issue, Ian Robertson wrote to ask how to use DeciGel without modifying every one of his disks (Letters, Volume 8, Issue 3). The answer that followed was correct, and since I switched to an MC68010 I have only found a few (less than 10) games that require the DeciGel patch to run. All others (about 80) load and run fine with no changes. I find that older V1.1 games are the common problem programs. Almost all of the better software

houses have written their newer programs and updates (V1.2 compatibility) with the MC68010 in mind. I have found two programs, however, that will not run: "Hacker" which, due to the form of copy protection used does not have a start-up sequence that can be modified, sorry; and "Transformer" from Commodore for which a fix follows.

I am not a user of Transformer and offer this fix/patch for those who feel they cannot upgrade because of losing Transformer. No longer need they suffer! If you were to install DeciGel on your Transformer disk and make the proper call to activate DeciGel when booting your machine, it would be lost as your Amiga resets itself when starting Transformer. So what do you do? A simple fix/patch to your Transformer program will solve the problem. This requires the use of a binary file editor – sorry, but Commodore did not supply one with your Amiga. Like DeciGel itself, the best place to obtain a file editor is the public domain. I prefer NewZap 3.0 available on Fish disk #58. The instructions that follow assume you are familiar with NewZap.

Make a backup copy of your Transformer disk and put the original back in its box. Put the copy in drive 'x' and start NewZap with the command:

1> newzap dfx:atl

Press return. Using the search function of NewZap (Amiga-z and Amiga-c) change all occurrences of these hex strings:

40c2, 40c4, 40c6, 40c7

to:

42c2, 42c4, 42c6, 42c7

This means that 40c2 will become 42c2, 40c4 becomes 42c4, etc. There are quite a few (I mean a lot!) of changes to be made, so relax, take your time and remember to save each disk block as you go. What you have for all this work is a copy of Transformer that will work with an MC68010. Please note that after Transformer is modified in this fashion, it will *not* run on an MC68000 machine. (That's why we made a copy, didn't we?)

While we are on the subject of Transformer, there is a patch available to make it run under AmigaDOS V1.2 in the public domain called "Trans12". It is not available on any of the Fish or Amicus disks, but should be available on services like People Link and Delphi, or check your local BBS or users' group for a copy. It is easy to use – just copy Trans12 to your copy of Transformer, CD to your Transformer disk and run Trans12. This will patch your Transformer for use under V1.2.

Daniel Schein, West Lawn, Pennsylvania

C-64 Keyboard Matrix: All is not well in the C-64 keyboard matrix. The program below visually represents which keys are being pressed. Type it in and save it before running; it turns off the normal keyscan. You can see the problem by pressing the Stop Key, Commodore Key, Q Key and Space Bar at the same time and then, with these held down, pressing the F1 key. The Cursor Down, F5 and F3 keys will come on as if they are being pressed. I stumbled across this on the way to a split keyboard routine. So far I've worked around it with a routine that uses the Right Shift and the Cursor Control keys for one player, with the Left Arrow and the One and Two keys for the other player. These keys are in mutually exclusive columns as seen by Data Port B, but this method limits the number of exclusive key closers to eight, makes finger placement awkward and an accidental key press can throw the whole scheme off. Any suggestions?

```
10 c=0: s=7: a=56320: b=a+1: cl=55296
20 dim m(7,7),x(7),y(7): r$=chr$(18): n$=chr$(146)
30 for i=. to s: x(i)=2↑i: y(i)=255-x(i)
40 for j=. to s: m(i,j)=(s-i)*40+(s-j)
50 next j,i
60 poke 53280,.: poke 53281,.: poke 646,1: poke 56333,127
70 print chr$(147);
80 print r$" q  2 ←1"n$"  stp q com spc 2 ctl ← 1"
90 print r$"/↑= ;*£"n$spc(12)"lsh hm"
100 print r$",@:.-lp+"
110 print r$"nokm0ji9"
120 print r$"vuhb8gy7"
130 print r$"xtfc6dr5"
140 print r$" esz4aw3"n$"  rsh"
150 print r$"[8 spcs]"n$"  crd f5 f3 f1 f7 crr rtn del"
160 for i=. to s: poke a,y(i): for j=. to s
170 c=1: if (peek(b) and x(j))=. then c=2
180 poke m(i,j),c: next j,i: goto 160
```

Owings Saffell, Eugene, Oregon

It is true that multiple simultaneous key-presses will often generate spurious keycodes. This isn't really a bug, though – just a side effect of the way the hardware is set up to map 64 keys onto 8 switches. For two-person use of the keyboard, it may be that the compromise solution you've already arrived at is about as good as you can do. Anyone else have ideas on this?

Clock Setting: This year I had an interface built for me which operates some lights in my home as well as the automatic sprinkling system for my yard and greenhouse. I wrote a program to run the interface in Basic on the 128 and it works just fine.

The problem I have is that, if there is a power cut, the program boots from disk but, of course, does not re-set the time of day. Is there a digital clock on the market that the 128 can read externally and then use to reset either of the two clocks resident in the computer? If not, *can* the 128 read an external clock and is there anyone who could provide me with the necessary hardware and a program to read the clock?

David Kuhn, Nanaimo, British Columbia

*Well, we're sure that there is **some** way to interface an external clock to the 128, but we don't remember ever having seen it done, commercially or otherwise. Readers?*

Transactor's Amiga Coverage: I see that you are starting to get flak about coverage of the Amiga. I vote with you. Face it, the 64 with its 8K operating system and 8K Basic interpreter, has been

pretty thoroughly dissected. Multiple editions of the *commented* ROMs are available, and even the Transactor's expertise is reflected increasingly in clever ways to use what is generally known, rather than mysteries revealed (though you still reveal more mysteries than anyone else). The Amiga is new, it's mysterious, and it's Commodore. I, too, cannot afford an Amiga, yet the idea that its coverage is inappropriate for the T. is preposterous.

Good luck in your endeavours, the Transactor is tops.

Tab Trepagnier, Kenner, Los Angeles

Well, thanks for the support, Tab, but as you now know we are no longer going to cover Amiga in this magazine. Still, you're right, we're running out of mysteries. Because of that, we'll probably be adjusting our format slightly in issues to come... along with the utilities and technical applications, we'll be adding more theoretical material (like the binary trees article in this issue) and reviews of selected products. Also – for those of you who have been clamouring for it – it looks as though we're finally going to get around to running a machine language subroutines column starting next issue. Anything else you'd like to see? Let us know.

"My heart leaps up when I can hail,
My Transactor in the mail. . . ":

1. Volume 7, Issue 5, page 10: "ScreenSave". Re: line 1200 – my verifizer says it equals HB. Your printed line says CK. I feel the error is in line 1200 as the program doesn't run – please adjudicate.

2. Volume 8, Issue 1, page 36: "Complex Numbers . . ." It seems that I need Vol 5 Issue 6 "new wedge", which I don't have. Is it possible to purchase a back copy from you, or can I get a photocopy from you or from another reader?

3. Regarding the Verifizer Codes in the left-hand column, could they be moved to the right-hand edge? It would be much more convenient.

Ernest Dorfman, Brooklyn, New York

Sorry, Ernest, but we still get CK for that Verifizer code. Is it possible you have a pair of transposed numbers in that DATA line, or possibly a period substituted for a comma? For other possibilities, take a look at the section on page 14 titled "Common by Comparison".

New Wedge is ***not*** *needed for the complex numbers program. It was referenced in the article as the place where an explanation of the wedge technique used in the program could be found.*

The idea of running the verifizer codes down the right hand edge of the listings is a good one. . . odd that it never occurred to us. We'll experiment with it between now and next issue and, if the typesetting works out, you should see some examples of it then. Thanks!

Accessing Curly Brackets with Speedscript: In response to the letter from John Francis (Letters, Volume 8, Issue 2) concerning accessing curly brackets with Speedscript: Assuming his terminal program will let him upload a file in true ASCII without translation, Speedscript has several features that will provide him with the curly brackets in a sequential file. First, he can put a formatting command of "Ctrl 3, a" as the first character of the file. This will translate everything into true ASCII. Then define any two uppercase characters as the ASCII codes for the curly brackets. I used shifted "+": "Ctrl 3, Shift + = 123", 123 being the decimal of $7B. Then do the same for 125 or $7D. I used Shifted "–". These control characters will show on the screen as reversed characters. Then whenever he needs the curly brackets, he just needs to hit "Ctrl 3, shift +" for left { and "Ctrl 3, shift –" for right }.

Speedscript then allows a file to be saved as a SEQ file in PETSCII (or ASCII if the above command is included) by hitting "Shift-Ctrl-P" and responding "D" to "Print to Disk". The reversed control characters are not saved since they are Print Format commands. Then he can just adjust his terminal program to upload with no translation and the correct ASCII codes will be sent. I checked this out by examining such a file I had printed to disk and sure enough, it contained no codes above $7F, and the curly brackets appeared as $7B and $7D.

I printed this letter on my Star NX-10 by locking my interface into Transparent mode and printing it as an ASCII file, rather than have the interface translate.

James Greek, New York, New York

And from what we can see, it worked great!

The Continuing BIT/.BYTE Saga: Jack Lothian's un-assembler (Transactor, Volume 6, Issue 4) is a fine program that has evoked several letters during the past year. I think this is primarily because the program is quite useful to many readers and is well structured and commented, enabling readers easily to understand its functioning. I like other readers have some comments and ideas to offer.

John Menke (Letters, Volume 6, Issue 5) pointed out the problem of disassembling the sequence $2C;$00;$A9 as BIT $00A9 – the Commodore Assembler will re-assemble this as BIT $A9, changing the BIT for $2C to $24, dropping the byte $00 completely and offsetting all the code that follows by one byte. Thomas Gurley (Letters, Volume 7, Issue 5) pointed out that this wasn't unique to the BIT instruction but was due to the absolute addressing mode. He observed that, if the addressing mode was one of the absolute modes (including absolute,X and absolute,Y) and the operand was less than 256, the assembler would change the addressing mode to the corresponding zero page mode and drop the byte $00. His suggested program changes to correct this problem caused the sequence $4C;$73;$00 to be disassembled as .BYTE 76;.BYTE 115;BYTE 0., rather than JMP $0073. The problem is that not all instructions wiht absolute addressing modes also have the corresponding zero page addressing mode. If they do, the zero page mode op code is 8 less than the absolute mode op code.

The following changes to the program as it originally appeared in Transactor should correct this problem.

```
1380 pp$ = "  ": ad$ = "  ": if t = 0 then 1405
1410 if n>0 and n<14 then pp$ = pp$ + n$ + " "
1435 if n = 14 then 1380
1730 q1 = q: gosub 2090: ad = q: gosub 2090
     : ad = ad + q*mh: p = p + 2: de = ad
1731 hn = 3: gosub 2180
1732 if ad<256 and mn$(q1) = mn$(q1-8) then n = 14: return
1792 if ad<256 and mn$(q1) = mn$(q1-8) then n = 14: return
1852 if ad<256 and mn$(q1) = mn$(q1-8) then n = 14: return
2015 rem convert absolute address less than 256
     to .byte instructions
2020 p$ = ad$ + ".byte " + str$(op) + ":.byte " + str$(ad)
     + ":.byte 0   ;*** was " + n$ + " ***"
2022 n = 0: gosub 2150: p = p + 1: return
```

These and other letters have suggested the preference for the user to determine whether he or she wanted the BIT ($2C) op code to be disassembled as BIT or .BYTE and some have suggested the proper method of implementing this option. It seems to me that this decision should be based on the context in which this instruction is found. Programmers use the $2C opcode for two distinct purposes. The first is its documented function of setting or clearing the N and Z flags. When the programmer uses the $2C op code for this purpose the preferable method is to disassemble $2C with the BIT op code.

The other use is the quasi-documented function of skipping 2 bytes when entered from the top and creating a side entry point immediately below the $2C byte, where a register is commonly set using the immediate addressing mode. When used for this purpose, it is preferable to disassemble $2C with the .BYTE pseudo op so that the side entry point is clearly shown. This also permits the disassembler to create a valid label for the entry point. Since the op code that immediately follows $2C is commonly for the immediate addressing mode, this assumption can be used as the criterion for determining whether to disassemble using either the .BYTE or BIT op codes. (Though this isn't a perfect criterion, it is an easily implemented "rule of thumb" that is valid probably 75% of the time at least.)

The suggested changes below should implement this change:

```
DELETE LINES 310 THRU 360
2090 if aa$<>"" then a$ = aa$:aa$ = "":goto 2094
2092 get#1,a$
2094 q = asc(a$ + n1$):return
2120 p = p + 1:n$ = mn$(q):n = md(q)
2122 if q<>44 then 2126
2124 get#1,aa$:q1 = asc(aa$ + n1$):if md(q1) = 2
     then n = 0:n$ = ".bit"
2126 return
```

My thanks go to Transactor for publishing many fine programs like the un-assembler and the letters from readers that they provoke.

Bill Taylor, Cupertino, California

ML EPROM Burner: Being an avid reader of Transactor, I know how technically minded the staff and readers are. This is why I have decided to ask you for advice concerning the following problem. I would like to write an ML program and burn it on an EPROM, which would then go in a cartridge. I have the technical and practical abilities to do it, I just don't know how to go about it. I've read different books on the subject of cartridges, but they are all very cryptic. I know that upon power-up the computer checks for a cartridge by comparing addresses $8004 to $8008 to the word 'CBM80'. If it matches, program control is transferred to the cartridge through the vector at $8000. However, the PROM I removed from a cartridge (Visible Solar System) doesn't have 'CBM80' at $8004, and the vector at $8000 jumps into the Basic input buffer. Also, all JMPs and JSRs have their target addresses start with $E (as in $E191) instead of $8. So basically this is what I want to know: What PROMs are generally used in cartridges? Why can't I find 'CBM80' at $8004? Am I reading it wrong? Should I just give up and buy a commercial cartridge-maker?

I also noticed extra room for another microcircuit in the cartridge. What could it be used for? And finally, are there any books explaining the operation of the Commodore 64 in detail? I would greatly appreciate this letter being published; even if you can't give me the answer, maybe another reader can help me out. I can be reached at the address below.

Patrick G. Demets, Box 1936, Grand Centre, Alberta T0A 1T0

Unfortunately, we do not have access to the Visible Solar System cartridge – pity, because we'd be interested in checking it out. We wonder if the cartridge is set up to use the Ultimax memory layout or one of the other less-used layouts charted in the Programmers' Reference Guide. The Ultimax layout does have cartridge ROM at $E000. The 'CBM80' ID is obviously not needed in this case because the hardware reset vector is within the address range of the cartridge itself. Any of you readers have more definite information on this?

On the question of books, there are lots that cover the 64 from a software viewpoint, but we assume that's not what you mean. Hardware information is a lot harder to find outside the Programmers' Reference Guide. Again – anyone have recommendations?

IEEE Interfaces for the C-64/128: You recently had a letter inquiring about IEEE interfaces for the C-64/128. I have used a couple of them and would like to give you my impressions of them.

First the MSD IEEE interface was mentioned. Well, it doesn't even make a very good door stop, unfortunately. It does work with plain vanilla Basic programs and very little else. Also there was a program that MSD gave out that did improve its performance. Even then it had a couple of big limitations and I quickly gave up trying to use that interface.

On to more useful interfaces for use with the C-64. I think it would be hard to find a better interface than the BusCard with the Basic 4.0 extensions, the ML monitor and the printer port in it. It also seems to be compatible with a lot of commercial software, though of course not all.

However, the BusCard does not work on the C-128 in 128 mode. For use with the C-128 I recommend the Quicksilver 128. I am currently

using that interface on my C-128. However, since Skyles replaces the Kernal ROM in the 128 with one of their own design, it is not compatible with CP/M. Skyles does sell a ROM switch board that you can install in your C-128 and have both ROMS available to you, but if you switch your original Kernal ROM back in, you still have to remove the Quicksilver cartridge in order to boot CP/M. As to drive compatibility with CP/M, I don't think that the higher capacity drives (8050, 8250, SFD1001) are compatible with Commodore's implementation of CP/M on the C-128.

One other thing: the SFD1001 working through the Quicksilver is as quick as if not quicker than the 1571. From my observations and the comments of other users, the data transfers at about the same speed. But the internal operations of the SFD1001 can run circles around the 1571.

I hope this helps your readers and provides a little background on these interfaces. I have used my IEEE interfaces with a MSD and a SFD1001 disk drive and am quite satisfied with my current set-up (C-128, 1571, MSD dual, and SFD1001).

Lyle Giese, Woodstock, Illinois

TransBloopers

Random Number Generation in ML, May 1987: Volume 7, Issue 6, pg. 28

Nothing wrong with program here. . . just a couple of random errors that crept into the text. Charlie Kluepfel of Bloomfield, NJ, writes:

. . .The article contains an inconsistency which makes me distrust the values shown in the table of Figure 2 (page 28). The diagram shows taps from 5 and 2, while the table shows taps from 5 and 1. Trying both out, in BASIC, shows that the diagram is correct and the table is wrong. Using taps at 5 and 1 gives a cycle length of 7; 5 and 2 gives the maximum, 31. Thus, how can we be sure the other entries on the table are correct?

By the way, the number '1021' in the right hand column of page 29 should be 10^{21} (or 10↑21). The former is not very "astronomical".

Our humblest apologies for this one, Charlie. We're also sorry we didn't get to this sooner, but the original manuscript for this article went into hibernation while we effected our "big move" and only just turned up again the other day.

After checking the manuscript, the taps shown for this spot in the table are indeed at **5 and 2**. Following your suggestion, we decided to check the rest of the table too, but found no other errors. The number 10↑21 checks out too, but c'mon Charlie. . . the difference is *only* 1,000,000,000,000,000,098,979 (1 sextillion, 98. . .).

And thanks for writing! We usually only find out about errors in programs. It's the meticulous types like yourself that allow others to go back and "update our product".

Common by Comparison

Everett E. Stone writes describing some troubles getting David Lathrop's "Compare" program (Transactor, Volume 8, Issue 1) to work – "I can not get the check sum of 26624 nor can I run the the program without adding another zero to the end of the DATA statements at line 7450."

I'll assume you're already aware that the checksum should be 523626 (26624 is the initial loop value), but that one's too easy. One of the most common and less obvious problems we find readers encounter with long listings of DATA statements is the omission of a comma. Usually this error gets detected because the comma is missing from between two numbers that, when concatenated, form a value greater than 255. However, if the comma is absent from between two numbers such as 2 and 44, the READ command "sees" a value of 244, which is okay in a POKE statement or CHR$ function. However, the program has just scooped up two numbers as one, and will run out of DATA before the READ loop is complete. Adding zeroes to the end of the DATA statements will satisfy the loop, but the entire program beyond the trouble spot will be "skewed" one byte.

Separating numbers with a period instead of a comma creates a similar situation. This one's equally hard to detect at first glance because the POKE command and CHR$ function simply ignore the fractional part and report no error.

Remember, machine language is an exact science: it's either right or it isn't. An ?OUT OF DATA error can indicate a missing comma, a missing DATA number, or even a missing line. The first thing to check is the counter value. If the loop is FOR J = 1 to 100, a "PRINT J" immediately after it's finished should return "101". If you get, for example, "99", you have two problems to go looking for.

Once the loop is ending correctly, it's possible the check sum will fall naturally into place. If it doesn't, there's a couple of other potential trouble spots you can look at before you conclude that the error is in the printed listing. If the variable "CH" is used to accumulate the sum (as is the case with most programs in Transactor), PRINT CH after the error report. If it contains a fractional part, chances are there's a period where there should be a comma. If it's off by an even 100 or 200, chances are you've keyed, for example, 49 instead of 149 or 249.

Lastly, the Verifizer entry checker will prevent almost any error, but it can be fooled. Verifizer uses a technique called Cyclic Redundancy Checksumming in arriving at its two-character code. However, the cycle length Chris chose was 4. Therefore, if two consecutive 3-digit numbers occur in a DATA line (e.g. ". . .208, 249, . . ."), entering ". . .249, 208, . . ." would produce the same code (comma is important, space is ignored). A cycle of 5 would virtually eliminate this possibility because only the oddest of DATA loaders ever use numbers with more than 3 digits. We've considered making a new Verifizer that checks for spaces within quotes and ignores remarks – perhaps we'll alter this too.

TeleColumn

Forums Get Face Lift

Some 3 months of preparation, a 19 hour online stint, several rounds of internal shuffling, and another two sessions of about 12 hours each, have led to the re-organization of the Data Libraries in CBMPRG and CBMCOM, the two forums we manage on CompuServe.

A little background: the forums originally kept all files and programs in one of 11 Data Libraries. Once in a Forum, one would enter "DL" at the main Function prompt followed by a number from 0 to 10, or simply DLn, where n was the DL number for those who didn't need to see the list of choices each time.

Some time ago, CompuServe extended their Forum software to allow 18 DLs, 0–17. It was suggested that DL 0 be reserved for Sysop activity, but DLs 1 through 17 could be titled any way deemed appropriate. Terrific! Now we could go and redistribute files in our DLs into more narrowly defined categories. CompuServe furnishes a number of utilities accessible only from Sysop accounts for doing just that, and it would be a simple matter of moving the files from one DL to another. Not so simple!

First we captured the titles and descriptions of every file from all 11 DLs in both CBMCOM and CBMPRG. There are thousands of files representing a few meg of code uploaded over the course of the last 4–5 years! Since none of this is Amiga software (that's all in the AMIGAFORUM DLs), a meg adds up to a LOT of programs. It didn't take long to realize that we'd bitten off a mouthful.

After a few weeks of evaluating the kinds of programs in there, we created a number of new categories. With 18 DLs per forum, minus 2 for the Sysop areas, we had 34 to work with. But we didn't want to use them all up – we wanted to leave room for future expansion should the need arise. Several times we were forced to discard, modify, combine, and create more new categories so that each and every file in the DLs would have a "place to go". We settled on the 31 titles below, and then determined which would go to CBMPRG and which in CBMCOM. We avoided, however, deciding the order of the DL titles, mainly because we thought this would change, and because we didn't have to at this time.

We knew in advance that many files in CBMCOM would eventually end up in CBMPRG, and vice versa. In fact, we decided that the C128 Terminal Programs DL in CBMPRG should go over to CBMCOM with the other telecomputing stuff. However, CompuServe has no remote Sysop utility for transferring files between Forums, short of downloading and re-uploading. So we designated a "transfer DL" in each forum that CompuServe could then transfer for us locally.

It took months to go through all the lists. Before actually printing up the lists, Nick trimmed down most of the descriptions to make them less unwieldy. Even then the total listing was around three inches thick. Each file was given a code from 1 to 31. Although several of the DL names weren't changed, we still went through each and every file looking for strays that would be better off in another category. Quite often Nick and I would look at each other and one of us would say, "what's this doing here?", and promptly mark it to be moved to one of the other 30 categories.

Then Came The Day

We fired up a terminal program on the Amiga and another utility that allowed 50 function key definitions through use of the SHIFT, ALT and "Amiga" keys. The codes we gave each file were used as indirection to a command stored in a function key that would move the file to the appropriate DL. It wasn't until this point that we actually picked the order of our DL names as they would appear to the user. With CompuServe's 19-character limit on DL names, this wasn't always easy!

CBMPRG Data Libraries	CBMCOM Data Libraries
0 CBMPRG Sysops	0 CBMCOM Sysops
1 New Uploads Oct–>	1 New Uploads Oct–>
2 Programmer Utils	2 Help & Forum Utils
3 Assembler Utilities	3 News and Articles
4 Demo PRGs & Subrtns	4 Reviews and Demos
5 The Forth Zone	5 Science & Education
6 The C Language	6 Write, Print & Plot
7 Non–Resident Langs	7 Home and Business
8 C128 Mode Only!	8 General Databases
9 CP/M and GEOS	9 Special Databases
10 Sector–Level Utils	10 Speech Synthesis
11 File Conversion	11 Exotic Applications
12 DOS Assistant	12 CBTerm & Related
13 Needs 1571 or 1581	13 C–64 Terminals
14 Orphan Computers	14 C128 Terminals
15 (Future Expansion)	15 BBS Prgs (64 & 128)
16 Transactor BITS	16 (Future Expansion)
17 Transactor Programs	17 (Future Expansion)

So, for example, if a file were to be moved to category 27 ("File Conversion", DL11), we would hit ALT F7 which would send a "MOVE 11" command. Category 17 was "Assembler Utilities" (DL3), so SHIFT F7 would do a "MOVE 3".

When the target DL and the source DL were both in the one forum, that was the end of it. But if a file in one forum was being sent to the other, the function key was set up to send it to the transfer DL. We did CBMPRG first. Files with category codes of 2,

5, 7, 8, 9, 12, 13, etc., required hitting F2, F5, F7, F8, F9, SHIFT F2, SHIFT F3, etc., but all of these produced a "MOVE 15", since DL15 was the transfer DL for files emigrating from PRG to COM.

The whole idea was to do most of the thinking offline, a habit we've formed over the years that's become hard to break even though we aren't charged for our online time. By using this approach it wasn't necessary for *us* to convert our code to the DL number – the computer would do it. In retrospect it all worked well, but we made the operation so mindless that we spent most of the time waiting and adding to our "CompuServe wishlist" of Sysop utility functions.

Okay, let's get started. Whoops! First we'll have to determine the destiny of the files that have arrived over the past 3 months. No problem. . . they'll all appear first and we'll just decide on the fly until we get to the file that's first on our lists.

Ummm, maybe we'd better close access to the DLs. It would be okay to download programs while we worked, but if anyone uploaded a new program it would get gobbled up faster than hot dogs at a baseball game. Our apologies to anyone who got cut off – we didn't mean to slam the door.

Now for CBMCOM. The same approach was simply modified slightly such that the function keys were all re-defined with the individual MOVE commands to the DL numbers within COM, and all the fkeys corresponding to CBMPRG codes got a "MOVE 17" – the transfer DL for files going over to CBMPRG.

A quick note to CompuServe staff would have the two transfer DLs swapped by the day after tomorrow, and it was on to "phase *=*+1"

About 1 meg of files would cross paths during the swap. Once back online it was a simple matter to go through the files that had arrived from the neighboring forum and redistribute them among the local DLs.

ACK! What are these doing here?! Oh well, so a few files accidentally got hijacked. While redistributing in CBMCOM we discovered about 6 or 7 files that really should have stayed in PRG. Another quick little note to CIS staffers, and back they went. We'd like to thank Alex Sutcliffe of CompuServe's Forum Support Group for his time and patience.

So what's this all add up to? Well first of all. . . MAN! Does CompuServe ever have a mountain of programs available. Some of them aren't worth the ferrite they're stored on for more than about 3 people in the entire galaxy. Others are so old that it was only the fact that they're out of the high traffic areas that kept them from the deadly ERA command. We in fact ERAsed a LOT of old and decrepit programs that were just cluttering up the joint, and managed to eliminate a lot of duplicates and obsolete versions. But most important is the effect our re-categorizing will have on the task of finding a program.

Before, if a user asked us if we had a program for a specific purpose, our best answer was, "yes, we have one", followed often by, "try DL X, or Y, or Z", with an "I dunno" as a reluctant alternative.

Now, you can ask for a program and chances are good that we saw it in our travels, chances are better we'll know exactly where to look, and odds are even good that you'll find it yourself in a quarter the time it took before. Not only that, but there are several programs that simply would have never even been found because they were stored in places where it was unlikely anyone would even look.

The "New Uploads DL" will make it a lot easier to see what's new without the need to BROwse *all* the various DLs looking for recent stuff. Once a month this DL will be cleared out so that only the most recent of uploads will be present.

Next step? The upgraded Forum software also extends the number of Sub Topic names from 11 to 18. Once again, Sub Topic 0 is reserved for Sysop chat (and no, it's not so we can talk behind your backs – it's mostly for letting the Sysops and assistant Sysops know the status of uploads and other forum management activity). That leaves, again, 34 spots for Sub Topic titles, which we intend to enter and alter over the next few weeks.

Why did we tell you all this? Well, it's the Display Area thing again. This may sound like another excuse, but we decided that before we go making major additions, we ought to get our house in order first. The forums demand a good percentage of our time and adding a new wing to the complex means even more time. Therefore, it made sense to try and polish up the present routine so it could be more self-supporting when the time comes to concentrate on the new construction. For those who care, the list of online tasks in order are:

1. New Sub Topic titles
2. New DESC files for the DLs. When you enter a DL, type DESC at the main prompt. This seldom used feature will print a description of the DL and what it contains. However, with all the moving around and re-categorizing, the DESC files will virtually all need to be replaced.
3. New HELP files in CBMCOM DL2. We answer a lot of questions in the message bases of the two forums. Often the questions are repeated every week, and we enter a new answer each time. Sometimes the most recent answer contains details not included in the previous one, and vice versa. It would be better, therefore, to collect all the details for the common questions and compile them into comprehensive, all-inclusive Help files. Further, members could read this online, or capture and read it offline. The objective would be to make enough Help files to eliminate the basic questions, saving the members time, money, and sanity.
4. Transactor Programs to CBMPRG DL 16 and 17. Yes, we're a little behind on this too, but we're working on it. Once again, the articles that go into the Display Area will often reference programs in the Transactor DL. We want them all in there so we know for sure what the file name will be.
5. The Display Area. When this opens it will contain 5 years' worth of text totalling around 5 meg of characters. We want to be sure that the text we put online is exactly the same as what

was printed, but with the mistakes fixed, of course. To do this, we're "reverse typesetting" the articles. Over the years, every article from every issue has been stored on typesetter 8" floppies. Since most of what we publish is spell checked at the type shop, and then edited in one way or another, we're using the version on the typesetter disks as the source. Then we have to take all the typesetter codes out and transmit the raw text out of the typesetter onto disks that can be read with equipment that can Xmodem transmit the text into our DA work area on CompuServe. We've done about half so far and are picking up speed as we go.

Stick with us. . . we only wanna do this once, so we wanna do it right.

Common Sense Function Keys

It seems that defining the function keys from within the Common Sense terminal program is not as obvious as most would like. Hit C= and K followed by the function key you wish to alter. From there you may enter up to 80 characters including "Shift @" to terminate the line.

The default definitions are saved in a file called MAC.BOOT. This file is loaded when the program is first RUN. To save the new definitions, rename the file "MAC.BOOT" and use C= S to save the new keys in a new file with the same name.

+ + + A Minus Minus Minus

Hitting the "+" sign three times while connected with a Hayes (or a truly Hayes compatible) modem will allow the user to temporarily suspend online activity while performing some other task. Theoretically the carrier signal isn't lost, but we all know that isn't true with some "Hayes compatibles". However, real Hayes compats allow the user to change the character from "+" to virtually any other. ATS2 = 64 will, for example, change it to the @ symbol.

Why would you want to do this? Truthfully, I don't really know. But Hayes must have included it for a reason. Initially I thought it might be possible to bring down a BBS by making it send you back three plus signs followed by a pause. But most BBS programs reset after a lost carrier or time limit. Any other ideas?

RS232 Interfaces

The RS232 interface project we published two issues ago is now working great for everyone who built it (at least, everone we've talked to about it). About the biggest problem was the note on the schematic about grounding all unused pins. That really should have said (as the article text did) "all unused inputs" – ie. the unused pins that were "input pins" should have been grounded. . . the others left as N/C. However, as reported in this issue's Bloops section, it's best to leave all the unused pins N/C.

However, for some, there's an even bigger problem: N/T – No Time! I remember a time when I would build every TTL gizmo I could find a schematic for. Now I barely have time to plug in a soldering iron. Sound familiar?

Omnitronix Deluxe RS232 Interface – $49.95 (206) 624-4985
Jameco Electronics JE232CM – $39.95 (415) 592-8097
Aprotek Universal RS232 – $39.95 (800) 962-5800 or (805) 987-2454

Which one is best? Perhaps someone would like to put together a point-by-point comparison and send it in. The write-up could even include our do-it-yourselfer by Mark Farris.

Plus 4 Terminal With Xmodem

Higgyterm (written by my good friend Paul Higginbottom) is probably the most common terminal program among Plus 4 owners. But you left out Xmodem, Paul!

P/4 TERM is one solution. Contact Dennis Larson, 11982 Xeon Street NW, Coon Rapids, MN, 55433. His CompuServe account number is 75555,705.

1650 Ring Detect and Answer in BASIC

Gary Farmaner of Oakville, Ontario, has this for anyone writing software to support a 1650 or other "non auto-answer" modems.

A 1650 isn't really an "auto-answer" modem. All functions are handled in software.

You have to monitor a line in $DD01 (56577) to see whether a ring condition is present. If bit 3 is low (=0), a ring is in progress. Then put the modem online by setting bit 5 of 56577 high and check for a carrier via bit 4.

```
10 reg=56577
20 open 2,2,0,chr$(6): poke 56579,38: poke reg, peek(reg) and 223
. . .
1000 rem wait for ring
1010 if peek(reg) and 8 then 1010
1020 rem ring in progress, answer it!
1030 poke reg, peek(reg) or 32
1040 rem check for carrier
1050 for x=1 to 10000
1060 if peek(reg) and 16 then 1120
1070 next
1080 rem timeout on carrier detect
1090 poke reg, peek(reg) and 223: goto1010
1100 rem start bbs
1110 rem clean up for/next loop and drop into bbs
1120 x=10000: next
```

Here's what's happening:
Line 10 opens the RS232 channel and sets up the port
Line 1010 waits for bit 3 of reg to go low
Line 1030 forces the 1650 online
Line 1050–1070 is a countdown carrier detect
Line 1090 re-starts the ring detect if no carrier within 10000 checks. If there is a carrier, the loop exits prematurely to. . .
Line 1120 cleans up the loop and starts the program. ■

Fast String Search With Binary Trees

Herb Rose
Dale City, VA

. . .a "divide-and-conquer" approach to handling a list of data items. . .

*Herb Rose is an Engineering Manager at CONTEL ASC. areas, and is the author of the INFO*SHARE multi-user bulletin board system, ARC64 file archival program for the C-64, and MACRO-64, a macro assembler for the Commodore 64.*

Organizing Your Data.

The first step in writing a program is usually to list the data items the program will manipulate, and to organize them into suitable data structures. Related items are grouped into RECORDS, sort keys are identified so that individual records are easily found, and other decisions are made concerning the organization of the data. The complexity of the program is often determined by the organization of the data which it must manipulate. It is no surprise that programmers spend a great deal of time learning how to create and use data structures. Knowledge of data organization techniques can turn a monstrous programming job into a real snap.

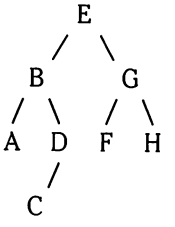
This is a short tutorial on one of the most useful and elegant data structures used by programmers, the Binary Tree. This article will explain what binary trees are, how to use them, and will present a BASIC program which creates and manipulates a binary tree which represents a string array.

What Are Binary Trees

A binary tree is simply a method of organizing data items within your program, in the same way that sorting is a way to organize data items within your program. A binary tree allows you to organize your data, insert and delete items, and search the list without moving any data within the list. A binary tree consists of NODES, which contain the data itself, and from each node there are BRANCHES. Each branch is actually a pointer to another node. When a rule for branching is applied to the tree, it becomes an efficient data sorting/searching tool. The rules we will be using for our binary trees are quite simple:

1. Duplicate nodes may exist.
2. Each node has 2 branches, which we will call left and right.
3. We will use the left branch to go to a lower or equal value
4. We will use the right branch to go to a higher value.

We now have the basis for a "divide-and-conquer" approach to handling a list of data items. As we traverse the binary tree searching for a particular data item, each branch we take brings us closer to the data we seek. The process is similar to using a binary search with a sorted list. Here is a sample binary tree which holds the letters A through H. Note that in some cases only one branch is used. In actual practice, each node has 2 branches, with unneeded branches marked in some way (usually a pointer value of 0). The node containing 'E' has arbitrarily been chosen as the first node in the tree (the BASE). All access to the tree will start at E.



Let's define some terms, using examples. A node is simply a tree entry which contains a value. Each letter entry in the tree above represents a node. Node 'E' is the base of the tree. It has no parent node, but it does have 2 branches ('B' and 'G'). 'B' is 'A's parent node. It is also 'D's parent node. 'A' is the left branch from node 'B'. 'D' is the right branch from node 'B'.

Remember that the picture above is simply the graphic representation of the binary tree. The data items are usually stored one or more arrays and the branches are kept as integer pointers to the next array entry (the index of the entry is usually kept, although in assembly language the actual ADDRESS of the next entry is sometimes kept as the pointer.

To set up a binary tree, you would use a program section similar to one of these (data consists of one string item and one integer item i.e. Name and AGE).

In BASIC:

```
100 dim a$(500),b(500): rem to hold data
110 dim l%(500),r%(500): rem left and right branches
```

In C:

```
struct tree {
   char mydata[datasize];
   int moredata;
   int leftptr;
   int rigthptr;
   } bintree[500];
```

Searching the Tree.

As stated above, searching a binary tree is similar to applying a binary search to an ordered list. In order to find any entry in the tree, we simply start at the base of the tree, and compare for a value that is equal to, less than, or greater than the value we are searching for. If the value at that level of the tree is equal to our search value, we are finished. If it is less than our search value, try the value on the right branch, else try the value on the left branch. This is because our rule states that we go to the left to find a lower value, and to the right to find a higher value. For example: we wish to find the value 'C' on the tree. Starting at the base, we find the value 'E'. 'C' is less than 'E', so follow the left branch. It contains the value 'B'. 'C' is greater than 'B', so use the right branch. It contains the value 'D'. 'C' is less than 'D', so use the left branch. There we find 'C', our search string, completing the search. We performed 4 compares to find a random element in an 8 entry list, not a bad performance. Now use that method to find 'G'. We only use 2 compares to find the element this time. That is a very good indication of the search speed of binary trees.

Inserting Tree Entries.

Insertion and deletion of entries is one of the most appealing aspects of binary trees. In order to add a new entry to the tree, simply follow the branches down as you would for a normal search, always going to the left when the string to be inserted is less than or equal to the value of the current node, and going to the right when the insert string is greater than the current node. When you find an unused branch, link your new entry there, and you are done. No data is moved within the array.

BASIC Programming Examples.

Here are several routines which demonstrate the techniques of searching, inserting entries into, and deleting entries from binary trees. These programming examples use a binary tree to access data in a string array named A$. Number arrays are used to keep track of the left and right branch pointers for each node. First is an insertion routine, which is used to add an entry to the binary tree. This routine does not add an entry to the string array; that must be done prior to calling this routine. What this routine does is insert the array entry A$(S) into the binary tree which describes array A$. L and R are the left and right pointer arrays. The variable S is loaded prior to calling this subroutine, and must contain the index of the array element being added to the tree. The essence of this routine, as with all routines which deal with a binary tree, is that we just keep going down the tree; using the branching rules described above, until we find what we are looking for. In this case, we are looking for an empty branch. When we find it, we will link this array entry into the tree by placing S into the empty branch pointer.

```
1000 k=0: rem k is the cur. node, start at base
1010 if a$(s)>a$(k) then 1050
1020 if l(k)<>0 then k=l(k): goto 1010
1030 rem ** this position empty, insert here **
1040 l(k)=s: return
1050 if r(k)<>0 then k=r(k): goto 1010
1060 r(k)=s: return
```

To build the tree from scratch given some number of records already in the arrays, use this short routine:

```
270 rem do not insert a$(0), we will use it as the
271 rem base of the tree.
272 rem the next line intializes l and r to 0
275 for s=0 to 5000: l(s)=0: r(s)=0: next s
277 rem now add each populated array element to the tree
280 for s=1 to 5000: if a$(s)<>"" then gosub 1000
290 next s
```

Next is the search routine, which searches A$ for an entry which matches T$. If one is found, its index is returned in S, else –1 is returned in S.

```
2005 s=-1: rem initialize s to show error return
2010 k=0 : rem initialize key to base of tree
2020 if a$(k)=t$ then s=k: return
2030 if t$>a$(k) then 2100
2040 if l(k)=0 then return
2050 k=l(k): goto 2020
2100 if r(k)=0 then return
2110 k=r(k): goto 2020
```

This routine is very similar in function to the insertion routine described above. Note that in lines 2040 and 2100, if the branch we wish to take is empty, we return –1 in S, indicating to the caller that T$ is not in the tree. In line 2020 we set S equal to the current node index and return, identifying the array entry which matches the search string by it's index.

Deletion of records is quite simple, but requires some explanation. If we wish to delete a record, we simply put a 0 in the appropriate branch pointer (L() or R()) of its parent node (the one just above it in the tree). That marks the branch as "unused". The astute reader will now wonder, "What if the record being deleted has branches?". It usually does. When we delete a node whose index is K, and which has branches, we create 2 smaller trees, whose bases are R(K) and L(K). All we must do is insert these trees back into the main tree (call the insert routine with S=L(K) and with S=R(K)). Here is the code to perform a deletion.

```
3000 rem d is the index of the record to delete
3001 rem first, find the parent node
3010 k=0:forj=0 to 1000
3020 if l(j)=d or r(j)=d then k=j:j=1000
3030 next j
3031 rem now k is the index to d's parent node
3040 if k=0 then return: rem not found
3050 if l(k)=d then l(k)=0
3060 if r(k)=d then r(k)=0
3061 rem the entry is now deleted from the tree
3062 rem now insert the branches
3070 if l(d)<>0 then s=l(d):gosub insert:l(d)=0
3080 if r(d)<>0 then s=r(d):gosub insert:r(d)=0
3090 return
```

This routine is rather incomplete, and messy. It will not allow you to delete the base node (index 0), and it searches for the parent node sequentially instead of using the tree structure. It does, however, illustrate the basic principle of deletion.

Binary Trees in C

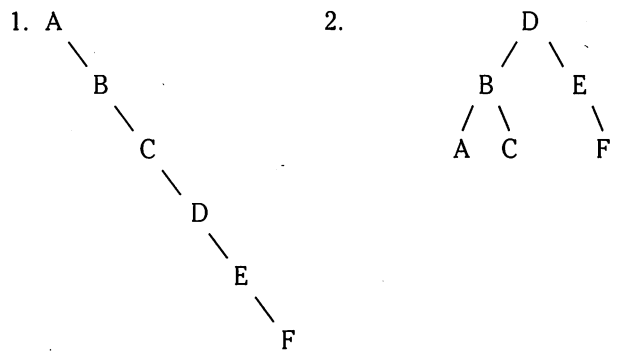
C and Pascal provide a method of simplifying binary tree manipulation. Both languages allow RECURSION, where a function or procedure can call itself. As an example – here is the search routine written in C, using the structure given above, using "moredata" as the search key :

```
search (key, curnode)
int key, curnode;
{
   if (key = = bintree[curnode].moredata)
      return key;
   if (key < bintree[curnode].moredata)
      return search(key, bintree[curnode].leftptr);
   else
      return search(key, bintree[curnode].rightptr);
}
```

There are ways of writing this routine in C which are more compact and perhaps more efficient, but this style demonstrates the techniques of recursion and tree searching very well.

Optimization

The efficiency of a binary tree can be measured in terms of the number of data comparisons needed to find a random element on the tree. This depends on the number of levels in the tree. The 2 trees shown below represent the same data, but are vastly different in shape and efficiency. There are 6 levels in tree #1, therefore a maximum of 6 comparisons would be needed to find an element at the lowest level of tree #1, while only 3 comparisons would be needed to find an element at the lowest level of tree #2.



Tree #2 is said to be balanced, while tree #1 is said to be unbalanced. There are algorithms which can be used to balance a tree that is unbalanced, but that subject is beyond the scope of this article. Here are some guidelines to follow when building binary trees that will help to keep them somewhat balanced :

1. Choose a base that is near the median point of the array, that is, half of the array elements are less than the base value, and half are greater.
2. Do not insert sorted data into the tree. Inserting the values A, B, C, D, E, and F into a tree (in that order) will produce a branch that looks like tree #1 above (try it!). Inserting D, B, E, A, C, and F will result in tree #2.

The Demo Program

The demonstration program is a BASIC routine that demonstrates the techniques for using binary trees, and shows you the speed advantage of incorporating binary trees into programs which must search lists of data. The program asks for the number of random strings to create, then it creates random strings and places them into an array. Each array entry is inserted into the binary tree as it is created. 10 random indexes are chosen, and the array is searched sequentially for the records. 10 more random indexes are chosen, and the search is performed using the binary tree. Next you are asked to enter a text string for the binary tree search to look for. This is a good demonstration of speed since the string you enter will probably not be found, so the search routine must descend all the way to the bottom levels of the tree. As with all sorting algorithms, the average amount of time saved saved per search is dependent on the number of array elements. If the array only has about 20 items in it, the binary tree search will take just about as long as the sequential search. For 1000 entries, the binary tree search is over 10 times as fast.

Binary trees are very general in nature, and will lend themselves to almost any application where fast data retrieval is necessary. They are great for symbol table manipulation in assemblers and compilers.

Binary Tree Demonstration Program

```
KB  100 rem binary tree demo program
DI  110 rem by herb rose
JM  120 input"size of string array ";n
AH  130 print"[CLR]binary tree demo"
GF  140 print"creating "n" random strings"
LM  150 dim aa$(n):dim l(n):dim r(n)
ML  160 sd=-ti:a=rnd(sd)
CL  170 fori=1ton
AB  180 print i"[UP]":n1=int(rnd(1)*10)+1:a$=""
FH  190 forj=1ton1:b$=chr$(int(rnd(1)*26+65))
        :a$=a$+b$:nextj
DC  200 aa$(i)=a$
OD  210 l(i)=0: r(i)=0
GC  220 ifi>1 then gosub 670
FM  230 next i
FJ  240 print"array created"
FL  250 print"sequential search for records"
MF  260 a=0
CI  270 for j=1to10
EL  280 n1=int(rnd(1)*n)+1
BC  290 b$=aa$(n1):t1=ti
```

```
IJ  300 i=1
PG  310 ifaa$(i)=b$then340
IB  320 i=i+1
JO  330 goto 310
AE  340 t2=ti
BL  350 s=int(((t2-t1)/60)*100)
PM  360 s=s/100
OF  370 a=a+s
MB  380 print "item #"n1"    "s"seconds"
IG  390 next j
FF  400 print "avg search time = "a/10
LC  410 print"binary tree search"
MP  420 a=0
CC  430 for j=1to10
EF  440 n1=int(rnd(1)*n)+1
BM  450 b$=aa$(n1):t1=ti
DJ  460 gosub 760
CM  470 t2=ti
FO  480 ifs=-1 then print"*";
KO  490 s=int(((t2-t1)/60)*100):s=s/100
AO  500 a=a+s
OJ  510 print "item #"n1"  "s"seconds"
KO  520 next j
HN  530 print "avg search time = "a/10
KD  540 b$=""
FE  550 input "string to find ";b$
EB  560 ifb$=""then end
DC  570 t1=ti
LA  580 gosub 760
KD  590 t2=ti
EO  600 ifs=-1 then print"not found";
CG  610 s=int(((t2-t1)/60)*100):s=s/100
LF  620 print "search time"s"seconds"
HC  630 goto 540
EP  640 :
HN  650 rem  the binary tree insert routine
IA  660 :
FG  670 k=1 : rem base is 1
MP  680 ifaa$(i)>aa$(k) then 710
MA  690 if l(k)<>0 then k=l(k):goto680
GN  700 l(k)=i:return
ID  710 if r(k)<>0 then k=r(k):goto680
GP  720 r(k)=i:return
OE  730 :
LA  740 rem  the binary tree search routine
CG  750 :
BJ  760 s=-1
CH  770 k=1
OC  780 ifaa$(k)=b$thens=k:return
JA  790 ifb$>aa$(k) then 820
AP  800 ifl(k)=0 then return
GI  810 k=l(k):goto 780
GB  820 ifr(k)=0 then return
CL  830 k=r(k):goto 780
```

Computers and Copyrights

Tony Romer
Edmonton, AB

. . . You have probably heard many stories of how poorly current copyright laws protect modern computer software. . .

Part I: Introduction

The purpose of this report is to explain copyrighting in Canada and the United States. Part I gives a background of copyrights in general. Part II explains how to copyright your software. Part III explains the legal aspects of copyrights. Part IV explains the use of Public Domain material (material not currently copyrighted).

Before explaining the copyright procedures, let's first make sure we fully understand the difference between software and hardware.

Software is a story of mystery and intrigue. It is a poem contrasting hope and despair. It is the portrait of the world through an artist's eye. It is a song of love.

Hardware is a pad of paper and a pencil. It is a book of pages with ink etchings. It is a canvas layered with pigmented oils of many colors. It is a plastic disc of uneven grooves, a turning circular platter, and a cut diamond.

If you can understand this, then you can understand the difference between software and hardware. Software is a purely non-physical collection of information. Hardware is a physical device (a chair, a desk, a pen). Most hardware has nothing to do with software, like chairs and desks. Some hardware is used to store or transcribe software – a pen, a typewriter, a record player, a sheet of paper.

So where are computers in all of this? Many people mistakenly relate software and hardware directly to computers. As you can tell from the above, software and hardware have nothing to do with computers. However, computers rely totally on software and hardware.

To computers, software is a collection of information. Hardware is the devices used to store and transcribe the software – printers, keyboards, monitors, disk drives, cassette drives, cassettes, cartridges. . . A subset of the software (a word processor, a video game, an operating system) may be used to provide information to control (or instruct) the hardware peripherals.

Now back to copyrights. Only software can be copyrighted. In fact, copyrights were created for exactly that purpose. That may seem hard to believe, since copyrights existed long before computers. You very probably have heard many stories (mostly true, unfortunately) of how poorly current copyright laws protect modern computer software.

Having read the above, though, you no doubt understand that software covers a wide range of information – books, poems, songs, paintings, photographs, documents, sound effects, moving pictures, and the list goes on. All of these forms of software are easily copyrightable even by the oldest of copyright laws.

Copyrights were formed to protect an individual's *expression* of an idea, concept, theme, algorithm or methodology.

Copyrights give copyright owners exclusive control over the reproduction and distribution of their copyrighted material.

It is the expression of an idea that is copyrightable, and not the idea itself. That is fundamental to copyrights. Splitting the fuzzy lines between an idea and a particular expression of an idea is a matter for the courts.

The oldest of copyright laws protects software as designed by the artist and any mechanical reproduction thereof. This pertained to both of the only two copyright conventions existing today – the Berne Convention and the UCC (Universal Copyright Convention).

The Berne Convention remains much the same today. The UCC has been widened broadly, and even has many special sections referring only to computers. Basically, the Berne Convention is used by the Commonwealth countries, like Canada and Australia. The UCC is used by the United States, Germany, Belgium. . .. Together the conventions cover virtually every country in the world.

Canada, and many countries, observe copyrights under either convention. However, you can only gain copyright by the Canadian Federal Government under the Berne convention. Fear not, though – you can gain copyrights in countries of which you are not a citizen, such as the US under the UCC.

With computers, the original copyright laws definitely allowed a programmer to copyright his or her source software and documentation. It is with object code that copyright problems arise. It seems courts just couldn't agree on whether the object code was a mechanical representation of the source code (like the grooves on a record, or a French translation of an English song), or a separate and distinct set of information not created by the author.

The US finally did something about the object code problem to protect the authors, and are continuing to update their laws. Changes in Canada's copyright laws are imminently upcoming. (*Just as they were a year ago, just as they were two years ago, and just as they will be five years from now.*)

What is not copyrightable? *Ideas, concepts, algorithms, themes, methodologies, and formulas (mathematical or otherwise) are not copyrightable*. Your only means of protection here is to keep your idea, algorithm, or formula a secret.

Films, audio tapes, video tapes, and phono records are not copyrightable! Read that sentence carefully. Those are all forms of hardware – their software contents may very well be copyrightable. It may seem quite obvious to you that you don't want to copyright an audio tape or a video tape – but remember never to use those words on a copyright form. Copyright offices are very sensitive about exact wording on their forms. *Never use words like 'idea', 'concept', 'methodology' and 'formula' on your copyright forms*. Use these words only in your documentation, which is part of the work to be copyrighted.

Like books, documentation can contain almost anything, and be copyrighted. So you can copyright a list of formulas, or a book of recipes. Thus no-one can exactly reproduce your list of formulas, or even print one of your recipes in the same form, without contravening your copyright. But they can use each and every formula or recipe. They can even produce their own list of the same formulas, or a book with the same recipes, provided the form in which they are presented is sufficiently different. As I said above, your only real protection here is secrecy.

I may set a camera on a tripod and take a photograph of a gorgeous waterfall against a beautiful sunset. I, and I alone, have the right to distribute copies of photos reproduced from the negatives at whatever price can be agreed to. On the other hand, you may have set your camera on the same tripod right after I stepped down, and have photographed the same scene. You also have the same rights to your photograph – no matter how similar it looks to mine. The scenery is not copyrightable – only the images as captured on film. You have yours, and I have mine.

A company comes up with a great idea for a video game. They decide to create a new caricature that eats dots and chases monsters with great sound effects. It could turn out to be a good idea, so they decide to use it. The game is created as a computer program, and is copyrighted. You see the game, and decide to create one for yourself. Have you violated a copyright law? This depends on the degree of similarity. Remember, ideas are not copyrightable. Therefore *anyone may produce a computer game of dot-eating, monster-chasing characters*. The moving pictures, sound effects, and code are copyrightable. If you haven't seen the code then you almost certainly cannot have violated that copyright. As for the rest, it must be very much alike to violate the copyright because, as I have said, anyone can use the same idea. Also remember this about your own programs – anyone can use your ideas without violating your copyrights.

What about titles? **Titles are not copyrightable.** Copyright offices do not even check these. You may, however, violate someone's registered trademark. Companies often register their titles as trademarks because copyright protection is not available. Copyright offices will not check for trademarks either – that is *your* duty. Common words will not be registered trademarks, like Blockade, or Breakout. Thus you may find very different software programs, fully copyrighted, with the same names.

If you would like to register your own title you will find it expensive ($300 or more). If you find a publisher for your program, chances are they will look after those details. What you want is to copyright it formally, so that the publisher cannot claim it as their own work. (You should beware of even notable companies with that regard – and copyrights are cheap to obtain.)

Copyrights are not designed only to protect the rich. Governments realize that many struggling artists and authors cannot afford high copyright costs. You can copyright your material for only $10 US in the States or $25 Cdn. in Canada. You can even copyright collections for the single cost in the US. Alternatively you could get a lawyer to copyright your works for you at $500 or more for each work.

Part 2: Obtaining A Copyright

You can obtain formal copyrights under the Berne convention in Canada or under the UCC in the States by requesting the appropriate forms for unpublished or published works from the corresponding governments. The forms will be sent to you free of charge.

In Canada, request Form 9 for an unpublished work, Form 10 for a published work. In the States, request form TX for computer works excluding audio and/or visual features. Request form PA (Performing Arts) if your work includes graphics and sound effects (arcade-type games, for example). Remember, you do not have to be a citizen of the country in which you choose to obtain a copyright (but a member of any country observing at least one of the two conventions, almost any country in the world).

Each government will forward additional information to help you fill out the forms. Wording on the forms must be very exact.

For copyright under the Berne convention write to:

The Copyright Office
Consumer and Corporate Affairs
Ottawa–Hull, Ontario
Canada K1A 0C9

For copyright under the UCC write to:

The Copyright Office
Library of Congress
Washington, D.C.
USA 20559

Or phone (202) 287-9100 and leave your request on the answering machine.

Copyrights in Canada cost $25 Cdn. for each work. Copyrights in the States cost $10 US for each work, or collection of works.

The forms will either have a line requesting the nature of the work, or the nature of the authorship of the work. If you are copyrighting non-audiovisual work (a utility program, an operating system, a compiler or a word-oriented game, for example), simply describe it as an *Artistic and Literary Work*. If your work includes graphics and sound effects, describe it as an *Artistic and Literary Audiovisual Work*. Alternatives, of course: Artistic and Literary Audio Work, Artistic and Literary Visual work. Do *not* say the work is an audiovisual work on cassette or on a floppy disk; leave the hardware medium out of your descriptions.

There is a place to show this as an anonymous or a pseudonymous work. In an anonymous work, you will not show any name of authorship on distributions of the work. Pseudonymous works will show a pen name on the works – Tony Romer is such a name.

Copyright forms allow you to show the true authorship of the work (authorship is not necessarily ownership). In fact, if you are using a formal copyright, you *must* show the true authorship.

You must show if the work was made for hire. If you created the work while salaried or commissioned by someone else then, as a work made for hire, it very probably belongs to the person or company that paid for it. Very probably – but not necessarily. Unless a formal agreement stating the nature of ownership has been made, the actual ownership may need to be settled by the courts. Regardless, show the work as a work made for hire if you have been paid to produce it.

The remaining sections of the forms should be matter of fact. You may explain the nature of the work more fully in the documentation accompanying the work. The forms were designed to contain all the required information for copyright – in fact, your form will be rejected if you give or show an attachment.

A title for the work must be shown. Titles do not have to be unique, and are not protected under copyrights. However, they cannot violate a registered trademark. It is your duty to ensure the title does not violate a registered trademark. The copyright office will not do this for you.

Works of authorship do not have to be formally copyrighted to be protected. However, formal copyrights are required under the UCC before a legal case can be instigated. Formal copyrights are the best way to establish the date of true authorship.

The Berne convention does not require any special affixations to a work to show that it is a copyrighted work. The UCC *does* require the work to be shown as copyrighted. Therefore, all works should be shown with the UCC markings to provide for copyright under the UCC, even if not formally applied for.

The UCC requires the work to be shown as Copyright at a specific date, and by a specific party. The only formal means of showing the copyright mark are: Copyright, Copr., or the letter c enclosed within a full circle. (Also the letter p enclosed in a circle for audio works only). Courts may also respect other symbols, such as (c) and (C). These are sometimes frowned upon, since they have not been officially adopted by the UCC. To my knowledge, however, they have never been rejected in a copyright case. The following, then are acceptable:

Copyright 1987 by Transactor
Copr. 1987 by Transactor Publishing Inc.
Copyright (C) by Transactor Publishing Inc. 1987
(C) 1987 by Transactor Publishing Inc.
(c) 1987 by Tony Romer

A court will allow copyrights without the ownership affixed – even if the work is totally anonymous (just "(C) 1987", for instance). However, the offices vastly prefer that the copyright holder be shown.

An author can allow one or more other parties to show a copyright holding of a work with their own names affixed without losing his or her full ownership of the true copyright.

In the United States, return one copy of your unpublished work, or two copies of your published work, with the forms. In Canada do not return any copies of an unpublished work with the forms, or two copies of a published work. If you follow these statements on copyrights in Canada, then you can see why I frown on Canadian copyrights – the office has no formal proof that you completed a work since it only received a title, and an extremely brief description of the work's nature.

You do not need to re–submit a previously copyrighted unpublished work for copyright after it has been published. A copyright exists for 28, 50 or 75 years depending on the convention copyrighted under, and renewal after initial copyright.

Obtaining a copyright does not fully establish an author as a true owner or creator of a work. It merely helps to establish the owner. Ultimately, if it ever comes to court, the legal process will decide the actual copyright holder. Which brings us to. . .

Part 3: Legal Aspects of Copyrights

Formal copyrights help to establish the ownership and date that a work has been completed. However, they don't guarantee full ownership by the holder. That is for the courts to decide. For example, John Doe may write a book anonymously, perhaps even including the proper copyright symbols. Long before the book becomes popular, John Smith applies to copyright the book, and successfully, though fraudulently, receives a copyright registration.

If John Doe can prove himself to be the true creator of the work (not made for hire) then he is indeed the true copyright holder. In the court's view, John Smith is far more likely to be the holder – since Mr. Smith has succeeded in obtaining a copyright on the work. However, if John Doe can produce a copy of the work from a sealed bank vault dated prior to John Smith's copyright application, the judge will probably find him to be the correct holder.

I hope this example illustrates why a formal copyright should be sought. John Doe will have a very difficult time proving his authorship unless he has a sealed copy dated such that it can be proved to precede someone else's copyright. Both alleged holders may be able to show any number of witnesses supporting their claims, so witnesses will not serve as well as having physical proof of ownership.

Many people ask if it is all right to copy copyrighted software for friends if no money changes hands. The answer is definitely NO! – it is not all right. A copyright gives the holder rights to distribute and regulate the distribution of copies. These rights have nothing to do with money.

People assume that software published in magazines is free to copy. Wrong again. In fact any work that is published is automatically copyrighted, and the magazine sends two copies of each magazine issue to the appropriate offices for formal rights. These companies are in the business of selling magazines – they give rights to each magazine holder to have or obtain a copy of the software on the basis that the magazine was purchased.

Remember that ideas, themes, and concepts are not copyrightable. If ideas were copyrightable, one person would probably hold a monopoly on murder mysteries, word processors, or computer operating systems. Ideas, no matter how simple or complex, are not copyrightable. It is only the full expression of the idea that is copyrightable – not the idea itself.

It is quite possible to create software that expresses someone else's ideas in apparently the same form without violating their rights. For example, you could completely imitate someone else's word processor, providing the code used is different. Similarly, someone else can do the same with *your* ideas.

An imitator should be aware, however, that portions of the work imitated may violate artistic computer designs, which are now copyrightable with other material under the Performing Arts. Closely imitating a screen design would be a copyright infringement.

Combatting computer theft has been a difficult and delicate issue. There have been some successful cases, though. Apple has won a number of cases in Australia protecting their ROM (Read Only Memory) software. A software publishing company successfully sued a magazine publishing company for a very large sum for publishing their methods of software protection under the guise of methods for programmers to protect themselves.

This supports one of my contentions: any article giving details of software protection only serves to help software pirates. A protection scheme becomes useless when anyone can find the scheme on a magazine stand. At best, magazines should provide methods for a programmer to go about creating his or her own protection – not details of a currently used scheme itself.

(Please note: Software Protection is a totally separate issue from Software Copyrights. Under a good protection scheme, the software should be fully copyable, and should not damage the user's hardware or software in any way. A legal owner of any software product has the right to make back-ups for as long as the original copy purchased is owned. The courts have made this legal right of software owners to make back-ups quite clear.)

Musical compositions are also copyrightable. Since much of the computer music supposedly in the public domain consists of versions of compositions actually created by other artists, such music almost certainly violates their rights – it is a mechanical reproduction of their work.

Of note here is a successful lawsuit made on behalf of the Beatles a few years ago. A company published a concert by the Beatles on video tape which, indeed, was not copyright and belonged to the public domain. However, the tape included the music – which *was* copyright, and therefore the copyright was violated. (In other words, they could legally only have distributed the tape without any sound.)

People work hard to produce their work. It may turn out to be quite good and marketable, or quite poor. The quality of the work is inconsequential to the copyright. These rights should be respected. If the artist is actually good enough to have the product marketed, then the royalties will encourage further work, and generally the quality of the work will increase because of the monetary rewards.

What do you think would have happened to your favourite rock bands and authors if everyone was freely copying their works? There probably would not have been much work left to copy. Would Arthur C. Clarke have written 2010 if he didn't receive any money for 2001? You will have to ask him.

Part 4: Public Domain Software

Public Domain Software is software for which copyrights are not currently in effect. In other words, Public Domain material is material that you can copy and distribute freely to your friends.

"Night of the Living Dead" and "It's A Wonderful Life" are Public Domain movies. Computer user clubs often have thousands of Public Domain games, utilities, and other software.

Most Public Domain material is so because that was the author's intent. However, sometimes material is not properly protected and falls into the Public Domain against the author's wishes. Once an item has fallen into the Public Domain, it is there to stay. However, copyrighted material is often distributed with Public Domain material by mistake – the copyright is still in force, and the distributor could be sued for its illegal distribution. Therefore Public Domain distributors generally try to avoid distributing copyrighted material without permission to do so.

There is nothing wrong with selling Public Domain material for a profit. Some companies stress that they are providing the software free and only charging for the service. Actually that is not necessary, and they could charge anything they wanted for the service anyway. Any notices restricting the buyer from distributing the software themselves is without force for true Public Domain material – anyone can distribute it, no matter how it was obtained.

It is actually the competition that keeps the cost of Public Domain material low. You should expect, though, to at least be paying a small amount for the equipment used in the copying process, the time used for the copying, and the shipping and handling charges. Some places may have volunteers doing the copying, but it probably would not last – it is rather boring work for a non-paying job. Therefore you should expect to pay for their time.

Because authors do not gain any monetary rewards for Public Domain material, you will find most of it rather simple software. You will need to sift through a lot of low quality software to find the real gems – and some of it will be quite good, like the movies mentioned above. You will find a lot of it without any instructions, and perhaps plagued with errors – well, what do you expect for such low cost software?

I am aware of a company distributing Public Domain software blended in a collection of copyrighted software all under the guise of a single copyright. This is probably fraudulent, since most of the software is Public Domain. Public Domain software cannot be copyrighted, and a court will probably take a dim view on the copyrightable software since it is indistinguishable from the non-copyrightable software.

You should be aware that some Public Domain distributors may legally include copyrighted works with their material, providing the author has allowed such distribution. This software will be clearly marked as copyrighted – you cannot distribute copies of it yourself without obtaining permission from the copyright holder.

Postscript

At the time that this article is undergoing final preparation for printing, efforts by the Canadian government to upgrade our current copyright laws to better protect computer software continue. While any changes in this regard are welcome (and overdue), I would like once again to emphasize that Canadians can copyright their material under the UCC by sending copies of their documents and software to the Copyright office in Washington, DC. Canada, and many other countries recognize copyrights under the UCC in addition to copyrights placed under the Berne convention. If you choose not to formally copyright your material, you should at least place copyright notices on your software where it will be clearly visible (Copyright 1987 by John Doe, for example).

Matrix Mathematics for the Commodore 64

Don Currie
Maidstone, ON

. . .Sometimes the things that were easy to learn are often the most time consuming. . .

Mathematics is the language of the sciences, but sometimes the language can get in the way of the solution. Yes, as powerful as math is, sometimes it can be too tedious to appreciate its power.

Sometimes the things that were easy to learn are often the most time consuming, such as matrix multiplication or the solution to a system of linear equations. That is why the following programs were written to allow a person to calculate the answers quicker and easier than traditional pencil and paper methods.

Matrix Multiplication

The product A*B of a matrix A with 'm' rows and 'p' columns and a matrix B with 'p' rows and 'n' columns is defined as C with 'm' rows and 'n' columns where c_{ij} (an element in C, with i representing the row and j the column of a specific element in the matrix) is defined as:

$$c_{ij} = a_{i1}*b_{1j} + a_{i2}*b_{2j} + \ldots + a_{ip}*b_{pj}$$

From the definition, the number of columns in A must be equal to the number of rows in B or else the definition does not hold true.

In any case, m, n and p can be any size value, but a practical limit for the computer is 20. If the maximum values are used for m, n and p, the memory usage on the Commodore 64 would be 8400 bytes at 7 bytes per number.

In the program, lines 100 to 390 serve to input the number of rows and columns of matrix A and B, do error checking on the values, and then the actual entry of the values into the matrix. The program will ask for the value in a specific row and column using the a_{ij} form for both matrices. Lines 410 to 600 are responsible for calculating the matrix product and printing the values in the new matrix. The values are printed in columns, one column at a time starting from left to right.

Solving N Linear Simultaneous Equations

Solving linear equations by hand can be a chore even with just a few linear equations. A technique often used to find the point where the lines represented by your equations intersect is Jacobi Iteration, which can approximate a system to any given tolerance.

In order to solve the system of n variables, you need n equations to get an actual numeric solution. In order to show you how to input the values into the program, an example will be used.

The computer will first ask for the number of variable you wish to solve for. For our case it is 2. The computer will then ask for the system of equations to be entered. Say our equations are:

$$1x + 2y = 3$$
$$4x + 5y = 6$$

We then take all the values and place them in a 2 rows and 3 columns matrix as

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

and input the values when the computer asks for the specific element in the matrix. The program will then ask for the tolerance the system is to be solved for - this value controls the accuracy of your approximation. For most situations, .001 is adequate. The next question is the number of iterations you wish to try to obtain the solution of the system. The greater the number of tries, the more chance there is of obtaining the solution to your specified tolerance.

The computer will then print the answer in column form for your system.

The Jacobi method is good for any number of equations, but the limit for this program is set at 20.

In the program, lines 100–240 are responsible for getting the number of equations and the matrix values from the user. Lines 250–360 check to see that the matrix was entered with no zeros in its principle diagonal. (The principle diagonal starts at the upper left and ends at the second last column at the bottom of the matrix.) If the matrix does have a zero in the principle diagonal, the program then seeks to arrange the matrix to have no zeros in the principle diagonal. Lines 370–390 get the tolerance and the number of iterations from the user. Lines 400–470 arrange the equations so the principle diagonal is as large as possible (a condition for the Jacobi method to work). Lines 480–700 perform the actual iteration work, and line 660 jumps to the answer output routine when the tolerance is met. Line 700 jumps to an error message when the system does not converge to your tolerance after the stated number of tries.

Conclusion

Matrix multiplication and solving linear equations occur in the sciences and engineering fields quite often in areas as diverse as special relativity to basic electrical circuit analysis. While the programs can be run without understanding how to perform the operations by hand, I encourage the reader to learn about their many applications and the theories and specific properties behind both operations.

Matrix Multiplier

```
BA  100 print"matrix multiplication":print
MD  110 print"this program finds the product of a"
OB  120 print"set of two matrices having up to"
CP  130 print"20*20 dimensions"
FJ  140 print"[DN RVS]matrix A:"
DL  150 print"how many rows and how many columns ?"
IE  160 input"  rows";r1
LC  170 input"  columns";c1
BM  180 print"[DN RVS]matrix B:"
LN  190 print"how many rows and how many columns ?"
DH  200 input"  rows";r2
FF  210 input"  columns";c2
CF  220 if r1>20 or c1>20 or r2>20 or c2>20 then
        print"[RVS]matrices too large":end
HE  230 if c1<>r2 then print "[RVS]# of cols in A not
        equal to # of rows in B":end
OA  240 rem
AD  250 dim a(r1,c1)
GD  260 print
EN  270 for r=1 to r1: for c=1 to c1
HJ  280 print"enter matrix A value";r;c;
GK  290 input a(r,c)
LP  300 next c,r
CH  310 dim b(r2,c2)
CH  320 print
IB  330 for r=1 to r2: for c=1 to c2
GN  340 print"enter matrix B value";r;c;
FO  350 input b(r,c)
HD  360 next c,r
EK  370 print
JI  380 print"the resulting matrix will have:"
JJ  390 print r1"rows and"c2"columns"
CM  400 print
FG  410 print"multiplication begins"
BO  420 dim c(r1,c2)
BK  430 for r=1 to r1
GA  440 for c=1 to c2: cs=0
BJ  450 for u=1 to c1
BP  460 cs=a(r,u)*b(u,c)+cs
JN  470 next u
LE  480 c(r,c)=cs: next c,r
HL  490 next c
OO  500 next r
HA  510 for c=1 to c2: print
LP  520 for r=1 to r1
EK  530 print"  C  ";r;",";c;"="c(r,c)
GB  540 next r
GC  550 print:print"hit return to see next column"
BF  560 get a$:if a$<>chr$(13) then 560
HA  570 next c
KE  580 input "require check view (y/n)  n[3 lefts]";a$
GM  590 if a$="n" then end
NP  600 goto 510
```

Simultaneaous Linear Equation Solver

```
BJ  100 print"jacobi iteration":print
IM  110 print"this program finds the solution to"
MC  120 print"a system of equations having up to
CO  130 print"20 variables with 20 equations."
KO  140 print:print"how many variables
CL  150 print"you wish to solve for";:input v
GC  160 if v=0 or v=1 then print "[RVS]too few":end
OB  170 if v>20 then print "[RVS]too many":end
FH  180 print "then there must be";v;"equations"
MC  190 dim m(v,v+1)
KP  200 print
IO  210 for r=1 to v: for c=1 to v+1
PH  220 print"enter matrix M value";r;c;
AK  230 input "  ";m(r,c)
PL  240 next c,r
AC  250 for k=1 to v
BD  260 if m(k,k)=0 then 290
DP  270 next k
PM  280 goto 370
FE  290 for j=1 to v
LA  300 if m(j,k)<>0 and m(k,j)<>0 then 330
IB  310 next j
ID  320 print:print"[RVS]sorry, jacobi iteration will not
        work on the system":end
EG  330 for t=1 to v+1
CN  340 ss=m(j,t): m(k,t)=ss
OF  350 next t
OM  360 next k: print
OB  370 print"to what tolerance do you wish
DH  380 print"the system to be evaluated to";
        :input"[2 spcs].001[6 left]";ot
ND  390 print "how many iterations";:input"[2 spcs]1000
        [6 left]";ri
AG  400 for j=1 to v: loc=j
DN  410 for i=j to v
BP  420 if m(i,j)>m(loc,j) then loc=i
NI  430 next i
EO  440 for z=1 to v+1
AC  450 s=m(loc,z): m(loc,z)=m(j,z): m(j,z)=s
ON  460 next z
IL  470 next j
GA  480 for k=1 to v
KI  490 s=-m(k,k): m(k,k)=0
LN  500 for c=1 to v+1
HE  510 m(k,c)=m(k,c)/s
FN  520 next c
NM  530 m(k,v+1)=-m(k,v+1)
BA  540 next k
MG  550 dim xo(v), nx(v)
HL  560 it=1
DG  570 for l=1 to v
FI  580 for t=1 to v
MI  590 xs=m(l,t)*xo(t)+xs
IF  600 next t
OA  610 nx(l)=xs+m(l,v+1):xs=0
EF  620 next l
EG  630 cs=0: for j=1 to v
AC  640 a=abs(xo(j)-nx(j)):cs=cs+a
MG  650 next j
AL  660 if cs/v<ot then 730
BM  670 for j=1 to v
CI  680 xo(j)=(nx(j)+xo(j))/2
EJ  690 next j
HO  700 it=it+1: if it<=ri then 570
HE  710 print"[RVS]no solution available
AF  720 print"[RVS]equations must not intersect.":end
MA  730 print
FC  740 for t=1 to v
DE  750 print"x";t;"=";nx(t)
KK  760 next: end
```

Read Infocom

Thomas W. Gurley
Wills Point, Texas

A Blind Walk Through The Black Forest

I am always most curious about that which is hidden, forbidden or secret. One of the first programs I bought for my Commodore 64 was Zork II, part of Infocom's celebrated original series of text adventures. Another was The Clone Machine, a program with features for examining raw sectors on disk. Somewhere between the two, I conceived a goal that was to take me down paths I never knew existed: I wanted to find out what words Zork II knew.

Quite innocently (never having used a "real computer" before) I began to search the tracks and sectors of the Zork II disk with The Clone Machine. To my surprise there were no secret words to be found! Suddenly it appeared that I was on a quest equal in magnitude to that of the search for the Holy Grail! Knowing absolutely no machine language, I started reading and studying.

I obtained an ML monitor and began to disassemble Zork II. Gradually, after literally months of reading, studying and disassembling, over and over and over, things began to make sense. I found the place where keyboard input occurs (even this was nearly a miracle because I knew nothing about the Kernal). Slowly, though, everything became easier, and eventually I discovered the answer to the question with which I had begun.

My reason for relating this history is to perhaps inspire others to "dig in". I now feel quite comfortable with assembly language, and I truly owe a debt of gratitude to the authors of Zork II. What began as a simple challenge turned out to be an almost total learning experience. The code in the Infocom programs is very well written, and has taught me a lot about programming technique.

As an aside, I should state that my delving into the labyrinth of Infocom was not for the purpose of piracy. In fact, the program that follows will not aid in such activity – it will merely give you an opportunity to study more closely an Infocom program you already own, and perhaps to help the bedraggled adventurer find his way.

Infocom Program Structure

Many of the Infocom games share a common structure. In fact, the first 8000 bytes or so are identical except for a few strings. The Zork I machine language driver works fine with the Sorcerer disk or with Infidel.

I do not have access to all of the Infocom games, so you will just have to test this program for yourself on a particular game. I know it works with the Zork series, Infidel, Planetfall, Enchanter and Sorcerer. It probably works on most or all of the others as well.

I will most often refer to Zork II, because that is the program with which I am most familiar. It is a long program, so I won't describe all its intricacies here. As far as my project is concerned, the first break came while I was actually playing the game. I noticed that when I typed a word Zork knew, it responded with disk drive activity or a phrase of some sort. If I entered a word Zork did not recognize, it responded "I do not know the word 'junket'", for example. There was no delay. That meant that all the words must be in memory all the time. But when I scanned memory, I found much the same thing I found on disk – no words.

Digging yet deeper into the code, I found a (to me) complicated character manipulation subroutine. First, the initial six characters of a word input from the keyboard are reduced to 5 bits apiece, then stored in addresses $5E–$63. The characters are then sent through the manipulation subroutine. This takes the second character and mixes it with the first, ORs the result with the third character and stores the result in the position of the first character. The second character is ROLed with the first, putting any carry bits in the low nybble of the first. This manipulation of bits completely obliterates all of the first three characters. Much the same thing happens to the last three. Refer to the source code of the routine at the end of this article. You will notice that only the last two characters are left unchanged, but those two are not used by the next subroutine, which scans memory for the four mixed-up numbers in the previous four bytes.

Location Of The Words

I have found three dictionaries in Zork II. The first begins at $2A40 and is a 'quick-search' list of often-used words. The second begins at $3615 and is a list of responses which Zork uses to talk to you. These words will not be recognized when entered from the keyboard unless they appear also in one of the other dictionaries. The third begins with the letter "a" at $692D. This is the list of words Zork recognizes as valid words from the keyboard.

All entries in these three dictionaries are made of four bytes – corresponding to the four mixed-up bytes encoded as described above.

Reconstruction

So. . . the words are in memory all the time, and now we know that only four bytes are used. I thought all I would have to do was simply unROL and unASL those four bytes to reveal the original word. Was I ever wrong! Remember those last two characters? They are very important for arriving at the mixed-up numbers, but they are not available going in reverse. All you have from the dictionary is four out-of-order, mixed-up bytes.

I wrote a Basic program that ANDed and ORed and divided and added. It was a monster that took about twenty seconds to unROL and unASL any four bytes (even if they amounted to nonsense), and there are several thousand bytes in the three word-lists. This first program used for-next loops to generate trial values for those missing last two characters. Surprise! Real words began to scroll by. That is how I found "bla@k can>le", which I immediately recognized as "black candle". But was "zifrS;" a word or just gibberish?

I came up with Basic equivalents of ASL, ROL, ROR and even ASR, which is missing from the 6510 opcodes.

But I wasn't looking for a copper grail.

Keep in mind that several months have passed since the beginning of this story. Several months of 20-hour days with 14-16 of those hours at home with toothpicks jabbed under the eyelids. By now, there was no turning back. Out of sheer frustration, I did put aside the project and I actually *played* Enchanter. Every time I played I wondered what secret words it knew that I did not know it knew.

I took out the long, slow Basic program once again, and used it on Enchanter. That is how I found "xyzzy" (at $7C74) and "antharion" and "zifmia". I ordered one of the "clue" books, but that only helped in solving the game. It did nothing for helping me know the words.

Is Machine Language The Answer?

Symass had come out by this time, and I was well enough acquainted with assembly language to write a simple program. I wrote a machine language program that created all possible combinations of the alphabet taken six at a time, and sent them forward through the manipulation subroutine. It then compared the result with four bytes from Zork II's word list, and upon no match created the next combination.

Perfect in theory - dismal failure in practice. Even at the lightning speeds of ML it took over thirty minutes to go from "aaaaaa" to "baaaaa", and on only the first entry in the dictionary. Well, let me tell you - I abandoned that approach after only the first try.

I almost gave up at this point because I had become convinced that all six characters must be known, and that the manipulation routine was a one-way street without the last two characters.

Like The Spaghetti Sauce - It's All In There

Then, like a blind man who suddenly sees, it dawned on me that *all six characters are contained in those four bytes*. Indeed, each pair of consecutive bytes in the dictionary contains three characters. From the moment I realized this until the secret words were scrolling by on the screen was less than eight hours.

Looking Back

In retrospect, it is obvious that the authors of Zork II were not only "hiding" the words from prying eyes, they were cramming three bytes into two, thus allowing more words for a more realistic response. Experienced programmers will probably laugh at my naivete, and so do I.

I am certain my program can be improved upon. Reconstruction of the original word from the mixed-up four bytes found in the three dictionaries is still not perfect. Following each valid word are several bytes that the parser uses to determine the response needed for that word. These bytes produce gibberish when unscrambled, sometimes producing real-looking words. You can ignore the gibberish or figure out how to skip over it. The parser uses some of these bytes as an offset to the next or previous entry. It adds or subtracts offsets, so finding a match is fast.

My program asks for a beginning address. RETURN will start you at $2A40 (10816 decimal). The increment for all entries is 2, 4 or 6, but since you do not know where the words are, an increment of 1 can be used. You can elect to increment by 2, changing the address to odd or even as needed.

The short dictionary at $2A40 is on even numbered addresses at an interval of two. You will immediately see valid words scroll past from this address.

The Final Program - The Holy Grail?

One would hardly think that so short and so simple a program is the result of so much time, effort and frustration. Even now, I'm not certain it is finished. Perhaps my grail is only gold-plated, and I welcome improvements. The words scroll past faster than I can write them down, so a machine language program isn't needed. Shift lock will halt the scroll. F1 allows changing the address and the increment.

A complete explanation of how the program works would be pointless and confusing without a listing of the parser source. I got mine with Un-assembler. Briefly, the original keyboard input is converted to a number from 6-32. Refer again to Figure 1. The lower 3 bits are shifted to the upper nybble (ASL 5 times). When the result is ORed with another number, the second number occupies the lower 5 bits. Thus one byte contains two characters - almost. Actually, the carry bits from the ASL are ROLed into the first character. My program tries to account for the carry bits and reconstructs the original six characters.

How To Actually See The Words

A reset switch is necessary. Just follow these steps:

1) Load and run an Infocom game.
2) At the first prompt ">", hit the reset.
3) Load and run "read infocom"
4) Memory is lowered to protect the dictionaries.
5) RETURN – increment 2.
6) F1 to change address and increment.

Between the dictionaries are sections of gibberish used as pointers and as storage area by the parser.

One final note: I chose to display six characters at once. Because of this, a short word or parts of longer words may appear on more than one line. I find it easier to recognize:

someth
ething
ing. . .

than to recognize:

som
eth
ing

but you can change line 350 and 380 to "FOR X=1 TO 3" if you prefer the second example.

Good luck and happy hunting.

Disassembly of the input manipulation subroutine (at $1ECF)

```
lda  $5f
asl
asl
asl
asl
rol  $5e
asl
rol  $5e
ldx  $5e
stx  $5f
ora  $60
sta  $5e
lda  $62
asl
asl
asl
asl
rol  $61
asl
rol  $61
ldx  $61
stx  $61   ;redundant, but it's there
ora  $63
sta  $60
lda  $61
ora  #$80
sta  $61
rts
```

```
EE 10 rem  read infocom
BJ 15 rem  1/3/87
PD 20 rem  thomas w. gurley
LK 25 rem  p.o. box 133
EA 30 rem  wills point, texas, 75169
HJ 35 :
MJ 40 :
JB 100 poke55,0: poke56,42: clr: rem lower top
      of basic
GK 110 aa=0: ab=0: ac=0: ad=0: m=0: n=0: o=0
      : nx=0: nn=0: diml(6): rem define variables
HO 120 print"[3 down][14 spaces]read infocom"
BD 130 print"[down]start ";: input"address";nx
PP 140 print"increment  1    2": input nn: if nn<1 or
      nn>2 then 140
DK 150 if nx=0 then nx=10816: rem start of short
      quick search word list
JN 160 if peek(197)=4 then print"address is "nx
      : goto130: rem change address with f1
OD 170 aa=peek(nx): ab=peek(nx+1)
      : ac=peek(nx+2): ad=peek(nx+3)
OH 180 m=0: n=0: o=0
ME 190 rem  95 94 97 96[29 spaces]aa ab ac ad
OM 200 l(1)=int((aa-20)/4)+64: rem the '+64' and
      '+59' restore alphabet position
CJ 210 l(3)=(ab and 31)+59: rem 'and 31' drops top
      three bits
FJ 220 m=int((ab and 240)/32)+59: rem 'and 240'
      takes top bits only
PK 230 if (aa and 1)=1 then n=8  : rem restore carry
      on second 'rol'
LM 240 if (aa and 2)=2 then o=16: rem restore carry
      on first 'rol'
DE 250 l(2)=m+n+o: rem add carries back in
AN 260 rem second half
PJ 270 m=0: n=0: o=0: rem cancel carries
EL 280 l(4)=int((ac-20)/4)+64
CB 290 l(6)=(ad and 31)+59
OE 300 m=int((ad and 240)/32)+59
DA 310 if (ac and 1)=1 then n=8
KK 320 if (ac and 2)=2 then o=16
MH 330 l(5)=m+n+0
GJ 340 print: print nx
JH 350 for x=1 to 6: a=l(x) and 223: if a<65 or a>90
      then a=46: rem '.' = invalid character
JJ 360 rem the 'and' 223 clears bit 5 to make lower
      case character
MJ 370 l(x)=a: next
NM 380 for x=1 to 6: print chr$(l(x));: next: print
PJ 390 if peek(654)=1 then 390: rem hold with shift
      lock
HO 400 nx=nx+nn: goto160
```

Interfacing Two Commodore 64's

Jack Bedard
Van Nuys, CA

Develop your program on one C64 and send it directly to another for testing!

How would you like to be able to assemble large machine language programs and data into 60K of memory without worrying about assembling on top of the assembler itself or the source code? Or how about keeping your machine undisturbed when the program you're testing crashes? Cross-assembling – assembling from one C-64 to another – is the way to do it. Code is developed and assembled on the *source* machine, and sent to a *target* machine to be tested. With the addition of the cable interface and software about to be described, very large programs can be developed and tested more easily.

Two complete C-64 systems are needed; 2 C-64's, 2 monitors and 2 storage devices (tape or disk). Also needed to make the cable:

1. 2 PC card edge connectors (dual row of 12 contact pairs, .156 X .145)
2. 2 protective covers for above connectors
3. 3 feet of cable (5 multi-stranded, multi-colored wires; 22 or 24 gauge will do)
4. a soldering iron (and enough skill to pick it up by the correct end)
5. solder and soldering flux
6. wire strippers

(Total cost of 1–3 is less than $10)

We are going to connect these two C-64s with only three feet of wire. Without going into a lengthy discussion of telecommunications and the errors that can arise during data transmission, let me point out that we should be able to obtain accurate data transmission and a very high transmission rate (Baud rate) with only 3 feet of cable connecting these two computers. We're not asking for very much, just a system that behaves as if the two computers are separated by only a few feet. Also, it would be desirable to be able to create communication software that is efficient in terms of both time and space, i.e. fast execution and requiring little memory. If the software doesn't interfere with other open channels (printer or disk), that would be an added bonus.

A number of sources have suggested using the mock RS-232 system that is already set up in the operating system. I went this route initially and found that none of the requirements just listed was met. The RS-232 channel is limited to an error-plagued 1200 Baud which may be adequate for transmission of text to and from a BBS but not nearly so for the transfer of a ML program where one bit out of place could crash the system. Furthermore, Device #2 must be opened, which results in longer programming code and interferes with my commercial assembler (MAE).

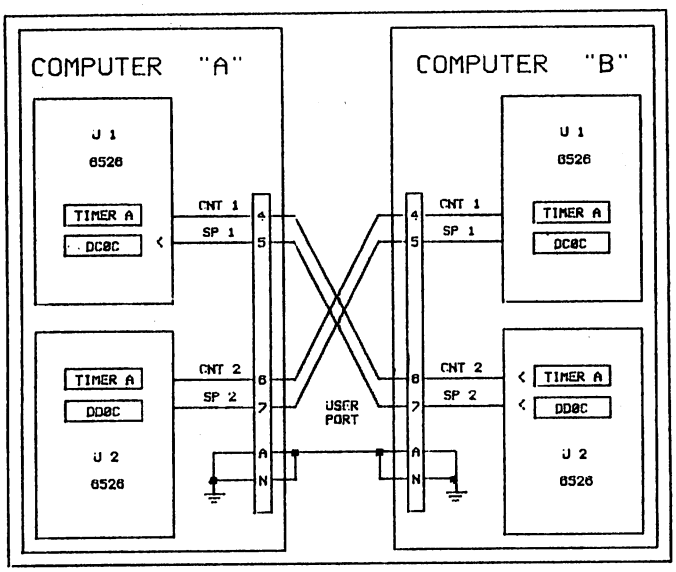
The Programmer's Reference Guide (page 432) gives a terse description of something called the "serial port" (not to be confused with the serial port that accesses the 1541 disk drive). It describes a hardware-implemented shift register that can be utilized by initializing two memory locations and TIMER A. Next, drop your data byte into the serial data register and the hardware will shift its bits one by one out on the SP pin – preceded (slightly) by a synchronizing signal (generated by TIMER A) on the CNT pin. When all the bits are gone, a particular bit in a control register is set to 1 (and if you so desire, an interrupt is generated) indicating another data byte can be sent. On the receiving end, just initialize those same two memory locations (ignore TIMER A) and keep checking (poll) the control register until that same particular bit is set to 1. Now go over to the serial data register and lift out the newly arrived data byte. It is that easy. If it wasn't for the fact that the shift register is wired to handle eight bits, Commodore might have incorporated it in the ROM routines that manage Device #2.

Let's look at some more details. There isn't just one serial port; there is one serial port on each 6526 chip and there are two 6526's on each C-64. I'm going to call the 6526's by their popular names: U1 and U2. (Also, I'm going to employ jargon used in technical writing about programming; when I say a bit is "set" that means it is set to 1; when I say a bit is "clear" that, of course, means it is set to 0.) U1's registers are in memory locations $DC00 to $DC0F and U2's are in locations $DD00 to $DD0F. The only locations we will be concerned about will be:

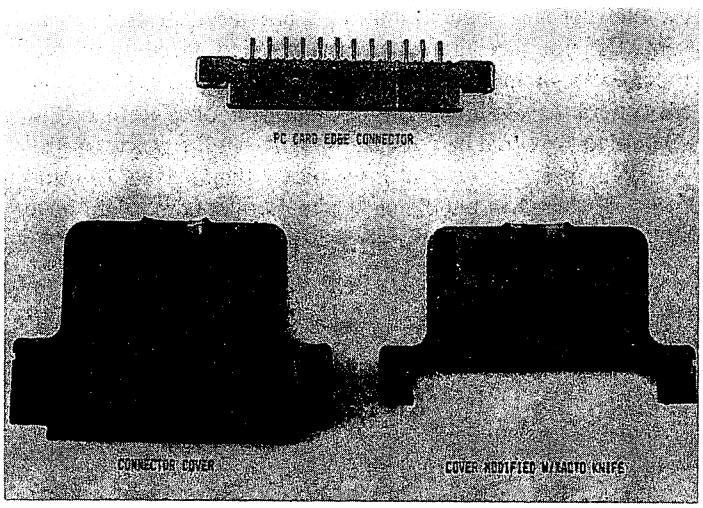
* $DC0C and $DD0C (the serial data ports)
* $DC0D and $DD0D (the interrupt control registers)
* $DC0E and $DD0E (TIMER A control registers)
* $DD04–5 (TIMER A on U2)

TIMER A on U1 is used by the system to generate interrupts (IRQ) every 1/60 second so housekeeping chores can be performed – reading the keyboard and that sort of thing. Not to worry, we need TIMER A only for output so we'll restrict ourselves to U2 for output and use U1 for input on each computer. (U2 to U2 would work also.) The accompanying diagram depicts the I/O scheme as well as the wiring arrangement. As shown, a dual simplex system is set up. It's sort of like a

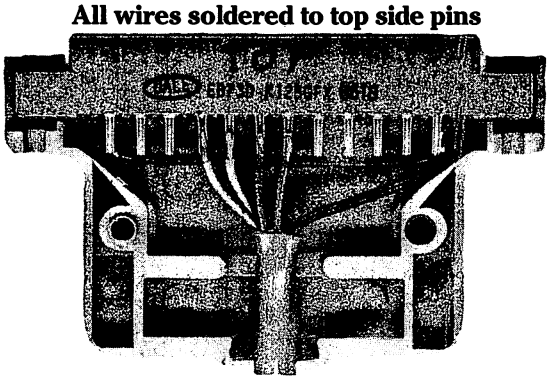
telephone hand-set where data goes out through the mouthpiece and comes in through speaker end. If error checking is desired, simply send and have the other C-64 return the data for verification.



Two C64s interfaced via their Serial Data Ports



All wires soldered to top side pins



The method we are going to implement is a polling one as opposed to an interrupt. (The RS-232 uses an interrupt method.) To output data we use U2:

Step 1. $DD0D: clear bit 7 and set bits 0 to 6; this will disable all interrupts on this chip.

Step 2. $DD04: give this low byte of TIMER A a very small value – like 4 (we do want a very high Baud rate). According to The Programmer's reference guide (PRG): "Data is shifted out on the SP pin at 1/2 the underflow rate of TIMER A." (If transmission errors occur, try increasing this value.) Then $DD05 gets a 0. I believe this low byte/high byte order is important.

Step 3. $DD0E: set bit 0 to start TIMER A; clear bit 3 to put TIMER A in the continuous mode; set bit 6 to select the output mode; clear all of the other bits.

Step 4. $DD0C: store your data to be transmitted here.

Step 5. $DD0D: read this location until bit 3 is set; this indicates the data has been transmitted; if you want, go to step 4 and do it again – steps 1–3 don't have to be repeated.

To input data on the other computer we use U1:

Step 1. $DC0D: see step 1 above.

Step 2. $DC0E: clear bit 6 without disturbing the other bits; this will select the input mode.

Step 3. $DC0D: read this location until bit 3 is set; this indicates that data has been received from the transmitting computer.

Step 4. $DD0C: read the data here.

Step 5. Process the data and go back to step 3 – steps 1 and 2 don't have to be repeated.

Step 6. When finished, re-enable the system IRQ by setting bits 0 and 7 in $DC0D.

Now, some comments on the programs:

Listing 1 and 2 are for users of the MAE assembler. The MAE receiver program is a mere 64 bytes in length and is tucked away in the top of page two. To run the program from a monitor enter:

.G 02AA

If you should want to run it from BASIC just change location $02D7 from $00 (BRK) to $60 (RTS):

POKE 727,96: SYS 682

The cursor will vanish and the keyboard will not function – except for the STOP key, which is used to terminate the program after MAE has finished assembling your ML program. You may now run your assembled program.

The MAE transmitter program has two functions. The first modifies the code of MAE itself so it will send the ML program to the other C-64 instead of storing it in memory. It does this by swapping the three bytes starting at $5FEB with the three bytes at $8514; the code at $5FEB then becomes "JSR $8517". The second routine is the actual transmitter; it sends the address of the byte about to be stored in the standard low/high byte order and then it sends the byte to be stored at that address.

The procedure to cross-assemble is this:

Step 1. In the receiving C-64 load and run your monitor program.
Step 2. load and run the MAE receiver program.
Step 3. In the transmitting C-64 load MAE and the MAE transmitter. Run the MAE.
Step 4. From within the assembler (MAE), enter "RUN $8500". This will swap the code as described above, thereby diverting the assembled code to the other machine.
Step 5. Assemble your program.
Step 6. After the assembly process is finished enter "RUN $8500" a second time. This restores MAE to its original state.

I have tested this software on several of my own programs with no problems and no appreciable increase in assembly time. I assembled a large program that was in six modules on disk; it ran in the other C-64 perfectly.

For further assistance in fathoming the workings of the serial port, I refer you to the pages of the PRG (432–434 and the fold-out schematic inside the back cover) and COMPUTE!'s Mapping The Commodore 64 (172–197).

One final note about the edge connectors: the wires to pins 4–7 are to be on top – label the connector to indicate that. I don't know what the results would be if the connectors were inserted into the user's port upside-down. Also, insert the connectors into the computers only with the power off.

Notes For PAL Users

For users of PAL or compatible assemblers, Listings 3 and 4 are provided. With the program in listing 3, you can send code directly from memory to the target computer. You lose the capability, as in the MAE version, of assembling very large programs that would overwrite your assembler or other memory-resident utilities, but for large programs you can send small portions of code at a time. You still have the advantage of not having to worry about losing your development environment when the code you're testing crashes. To use it, assemble your PAL program to memory as usual, then send it to the target machine with the command:

SYS 40850,<start address>,<end address>

The start and end addresses are in decimal, and tell the send routine which bytes to send. For example:

SYS 40850,49152,50200

The "receive" program in listing 4 operates exactly like the MAE program in listing 2, except that it is stored in the cassette buffer at 828 so that it doesn't conflict with other programs that use that area, like POWER. Use it from BASIC with a SYS 828. Press the STOP key to exit to BASIC after all code has been sent, then SYS to your program to test it.

Listing 1: This program, in MAE assembler format, will patch the MAE assembler so that when assembling it sends object code directly to the target computer through the serial cable.

```
0020 ; copyright 1986 jack bedard
0030 .
0040 ; output assembled code to serial data port
0050 .
0060 ; there are 2 separate routines here:
0070 ; the 1st (code.swap) modifies mae to transmit assembled code
0080 ; to a 2nd c-64 via s.r. port.
0090 ; it is activated with this command from mae 'ru $8500'
0100 .
0110 ; the 2nd sends the address to store (low/high)
0120 ; in the other c64 and the byte to store there.
0130 .
0140 mae.table .de $51          ;store address is in this table
0150 mod.adr   .de $5feb        ;my patch goes here
0160 u2.tima.lo .de $dd04
0170 u2.tima.hi .de $dd05
0180 u2.out    .de $dd0c        ;serial data port
0190 u2.icr    .de $dd0d        ;interrupt control register
0200 u2.cra    .de $dd0e        ;timer a control register
0210 output    .de %01000000 ;bit 6 in $dd0e
0220 shift.reg .de %00001000 ;bit 3 in $dd0d
0230 disabl.all .de %01111111 ;0 in 7 causes the 1's to disable those bits
0240 timer.a   .de %00000001 ;bit 0 in $dd0e
0250 .
0260 baud      .de $04          ;the baud rate prescaler
0270 .
0280    .os
0290    .ba $8500
0300    .ce
0310 .
0320 code.swap                  ;patch 'jsr sendtodsp' into mae
0330    ldx #2
0340 mod.loop
0350    lda mod.adr,x
0360    pha
0370    lda mae.code.mod,x
0380    sta mod.adr,x
0390    pla
0400    sta mae.code.mod,x
0410    dex
0420    bpl mod.loop
0430    rts
0440 .
0450 mae.code.mod               ;patch to our output routine
0460    jsr send.to.sdp
0470 .
0480 send.to.sdp                ;send byte from stack (under return addr)
0490    sty save.y              ;to other 64.
0500    stx save.x
0510    pla
0520    sta ret.adr             ;save return address from stack
0530    pla
0540    sta ret.adr+1
0550 .
0560    lda mae.table,x         ;find address for byte to be sent
0570    sta data.out+2
0580    lda mae.table+1,x
0590    sta data.out+1
0600    pla
0610    sta data.out            ;byte to send (after address)
0620 .
0630    lda #disabl.all         ;set up interrupt control reg
0640    sta u2.icr
0650 .
0660    lda #baud               ;set up timer
0670    sta u2.tima.lo
0680    lda #0
0690    sta u2.tima.hi
0700 .
0710    lda #output+timer.a     ;set up timer control register
0720    sta u2.cra
0730 .
0740    ldx #2                  ;send the three bytes starting at dataout
0750    sei                     ;no interrupts, please
0760 out.data
0770    lda data.out,x
0780    sta u2.out              ;put the byte on the output port
0790 .
0800 still.sending
0810    lda u2.icr              ;wait until it has been sent
0820    and#shift.reg
0830    beqstill.sending
```

```
0840    dex                     ;send the next one
0850    bpl out.data
0860    cli                     ;all sent
0870 .
0880    lda ret.adr+1           ;put return address back on the stack
0890    pha
0900    lda ret.adr
0910    pha
0920    ldx save.x
0930    ldy save.y
0940    rts
0950 .
0960 save.x     .ds 1
0970 save.y     .ds 1
0980 ret.adr    .ds 2
0990 data.out   .ds 3
1000 .
1010    .en
```

Listing 2: MAE–format "receive" program. This program runs on the target computer and loads all code sent by the source machine. Just SYS 682, send the code from the source machine, and press the STOP key after the data has been sent.

```
0010 ; copyright 1986 jack bedard
0020 .
0030 ; receive assembled code via
0040 ; the serial data port.
0050 .
0060 ; receives address to store
0070 ; (low/high) and the byte
0080 ; to store there.
0090 .
0100 ptr        .de $fd
0110 u1.input   .de $dc0c
0120 u1.icr     .de $dc0d
0130 u1.cra     .de $dc0e
0140 update.91  .de $f6bc
0150 stop       .de $91
0160 .
0170 output     .de %01000000
0180 shift.reg  .de %00001000
0190 disabl.all .de %01111111
0200 enable     .de %10000000
0210 timer.a    .de %00000001
0220 .
0230    .ce
0240    .ba $02aa
0250    .os
0260 .
0270    tsx
0280    stx save.sp
0290 .
0300    lda #disabl.all         ; disable interrupts
0310    sta u1.icr
0320 .
0330    lda u1.cra
0340    and#$ff-output          ; clear bit 6 of cra. . . serial port input at
0350    sta u1.cra              ; external clock rate
0360 .
0370 main.loop
0380    jsr get.sdp             ; get address of data byte
0390    sta ptr
0400    jsr get.sdp
0410    sta ptr+1
0420    jsr get.sdp             ; get data byte
0430 .
0440    ldy #0
0450    sta (ptr),y             ; store data byte
0460    beqmain.loop
0470 exit
0480    lda #enable+timer.a
0490    sta u1.icr              ; re-enable interrupts, restore stack, quit
0500    ldx save.sp
0510    txs
0520    brk
0530 .
0540 get.sdp
0550    jsr update.91
0560    lda stop                ;check stop key
0570    bpl exit
0580 .
0590    lda u1.icr              ;wait for input char
0600    and#shift.reg
0610    beqget.sdp
0620    lda u1.input            ;get input char
0630    rts
0640 .
0650 save.sp    .ds 1
0660 .
0670    .en
```

Listing 3: PAL–format source code for program to send an area of memory from the source machine. This program is placed at the top of memory; to use it, SYS 40850,<start address>,<end address>. Use this to send the object code of a program to the target machine after you assemble it.

```
IA  1000 open1,8,1,"0:sendcode.obj"
HJ  1010 sys 700                    ;activate pal assembler
GJ  1020 * = $9f92                  ;top of memory (40850)
EO  1030 .opt o1                    ;output object file
NH  1031 ;
LB  1040 ; output code to serial data port
AJ  1050 ;
JE  1060 ;this routine can be used to send
DI  1070 ;your code to the target computer
EK  1080 ;after it has been assembled.
MO  1090 ;call it from your assembler
HI  1100 ;environment and pass it the start
FH  1110 ;and end addresses of the object
MC  1120 ;code, e.g. sys 40850,49152,50261
AO  1130 ;
GM  1140 ;the routine 'sendtosdp' sends the
JO  1150 ;address to store (low/high)
NB  1160 ;in the other c64 and the byte to
JP  1170 ;store there.
CB  1180 ;
LH  1190 u2timalo   =   $dd04
CH  1200 u2timahi   =   $dd05
FO  1210 u2out      =   $dd0c       ;ser data port
KC  1220 u2icr      =   $dd0d       ;interrupt ctrl
KI  1230 u2cra      =   $dd0e       ;timer a ctrl
LG  1240 output     =   %01000000   ;$dd0e bit 6
HH  1250 shiftreg   =   %00001000   ;$dd0d bit 3
HI  1260 disablall  =   %01111111   ;clr ints
FP  1270 timera     =   %00000001   ;$dd0e bit 0
GH  1280 ;
CL  1290 codeptr    =   $fb         ;ptr to byte to send
KN  1300 endcode    =   $fd         ;last byte to send
EJ  1310 ;
MH  1320 baud       =   $04         ;baud rate prescaler
IK  1330 ;
CL  1340 ;
MJ  1360 sendcode   =   *
NC  1370            jsr getparam    ;start addr in y,a
CJ  1380            sta codeptr+1
DC  1390            sty codeptr
IK  1400            jsr getparam    ;get end address
HF  1410            sta endcode+1
KN  1420            sty endcode
MA  1430 ;
HE  1440 sc1        =   *           ;send all bytes
FP  1450            ldy #0
FB  1460            lda (codeptr),y ;get next byte
JE  1470            jsr sendtosdp   ;send it to port
OC  1480            inc codeptr     ;point to next byte
DM  1490            bne sc2
MO  1500            inc codeptr+1
FI  1510 sc2        =   *           ;end when codeptr = endcode
HA  1520            lda codeptr
PO  1530            cmp endcode
CP  1540            bne sc1
OP  1550            lda codeptr+1
EP  1560            cmp endcode+1
AB  1570            bne sc1
IB  1580            rts
MK  1590 ;
GL  1600 ;
PD  1610 sendtosdp  =   *
HJ  1620 ;send 'codeptr' pointer user port,
NH  1630 ;followed by byte in .a
PI  1640            sta dataout     ;byte of code to send
NO  1650            lda codeptr     ;addr of byte to send
BC  1660            sta dataout+2
GH  1670            lda codeptr+1
DD  1680            sta dataout+1
AB  1690 ;
PG  1700            lda #disablall
ML  1710            sta u2icr       ;clear all interrupts
```

```
OC 1720 ;
HO 1730          lda  #baud        ;set up timer
KD 1740          sta  u2timalo
BM 1750          lda  #0
KC 1760          sta  u2timahi
AG 1770 ;
PL 1780          lda  #output+timera
LP 1790 ;set up timer control register
JL 1800          sta  u2cra
 I 1810 ;
HG 1820          ldx  #2
PD 1830 ;send 3 bytes starting at dataout
NM 1840          sei               ;no interrupts, please
CK 1850 outdata  =    *
 IP 1860         lda  dataout,x
HP 1870          sta  u2out        ;put byte on the port
OM 1880 ;
KH 1890          lda  #shiftreg
OM 1900 stillsdg =    *
NM 1910 ;wait until it has been sent
BC 1920          bit  u2icr
BA 1930          beq  stillsdg
NN 1940          dex               ;send the next one
DE 1950          bpl  outdata
HA 1960          cli               ;all 3 bytes sent
 IC 1970 ;
 IK 1980         rts
MD 1990 ;
KF 2020 ;
DF 2030 getparam =    *
AF 2040 ;skip comma, get argument and put
HF 2050 ;in y (low) and a (high)
CL 2060          jsr  $aefd
OH 2070          jsr  $ad8a
HH 2080          jmp  $b7f7
AK 2090 ;
BK 2091 ;
HJ 3000 dataout  *=*+3
 KI 3010 ;buffer for addr and byte to send
```

Listing 4: "Receive" code in PAL format, identical to the MAE program in Listing 2. This code is located in the cassette buffer starting at 828.

```
PD 1000 open1,8,1,"0:rcv.obj
DM 1010 sys 700
KA 1020 .opt o1
DH 1030 ; copyright 1986 jack bedard
 GI 1040 ;
 CI 1050 ; receive assembled code via
FN 1060 ; the serial data port.
EK 1070 ;
JH 1080 ; receives address to store
OB 1090 ; (low/high) and the byte
LG 1100 ; to store there.
MM 1110 ;
AC 1120 ptr      =    $fd
DE 1130 u1input  =    $dc0c
MM 1140 u1icr    =    $dc0d
PK
KG 1160 update91 =    $f6bc
DP 1170 stop     =    $91
CB 1180 ;
JD 1190 output   =    %01000000
MP 1200 shiftreg =    %00001000
HH 1210 disablall=    %01111111
EJ 1220 enable   =    %10000000
PN 1230 timera   =    %00000001
OE 1240 ;
DH 1250 *= 828   ;goes in cassette buffer
CG 1260 ;
DG 1261 ;
GA 1270          tsx               ;save stack ptr for
CL 1280          stx  savesp       ;clean exit later
 AI 1290 ;
PN 1300          lda  #disablall
KD 1310          sta  u1icr        ;disable interrupts
OJ 1320 ;
MG 1330          lda  u1cra        ;clear bit 6 of cra
HD 1340          and  #$ff-output
AN 1350 ; serial port input at
 ID 1360         sta  u1cra        ;external clock rate
AN 1370 u1cra    =    $dc0e
MK 1380 mainloop =    *
KD 1390          jsr  getsdp       ;get data byte addr
MA 1400          sta  ptr
KK 1410          jsr  getsdp
JP 1420          sta  ptr+1
MJ 1430          jsr  getsdp       ;get data byte
GB 1440 ;
FP 1450          ldy  #0
KC 1460          sta  (ptr),y      ;store data byte
MK 1470          beq  mainloop     ;loop forever
OD 1480 ;
PD 1481 ;
EE 1490 exit     =    *
AN 1500 ; re-enable interrupts,
PN 1510 ; restore stack and quit
JP 1520          lda  #enable+timera
AJ 1530          sta  u1icr
CP 1540          ldx  savesp
KA 1550          txs
EA 1560          rts
 IJ 1570 ;
 JJ 1571 ;
PC 1580 getsdp   =    *
CO 1590 ; get byte from serial data port
 AI 1600         jsr  update91     ;check stop key
CE 1610          lda  stop
 IB 1620         bpl  exit
EN 1630 ;
 JJ 1640         lda  u1icr        ;wait for input byte
KP 1650          and  #shiftreg
JA 1660          beq  getsdp       ;no byte; loop again
PD 1670          lda  u1input      ;read the byte
MH 1680          rts
AB 1690 ;
CP 1700 savesp   *=*+1
```

Listing 5: BASIC program to generate "sendcode.obj", the same code produced by assembling the PAL source in Listing 3. Load this program and SYS to it as explained in the article sidebar.

```
OL 100 rem  program generator for "sendcode.obj"
EO 110 n$="sendcode.obj": rem  name of program
LA 120 nd=103: sa=40850: ch=16244
```

For lines 130-260, use the standard generator program from page 5

```
AE 1000 data  32, 240, 159, 133, 252, 132, 251,  32
PB 1010 data 240, 159, 133, 254, 132, 253, 160,   0
PE 1020 data 177, 251,  32, 186, 159, 230, 251, 208
OO 1030 data   2, 230, 252, 165, 251, 197, 253, 208
DE 1040 data 237, 165, 252, 197, 254, 208, 231,  96
FG 1050 data 141, 249, 159, 165, 251, 141, 251, 159
GG 1060 data 165, 252, 141, 250, 159, 169, 127, 141
FF 1070 data  13, 221, 169,   4, 141,   4, 221, 169
 IF 1080 data  0, 141,   5, 221, 169,  65, 141,  14
CF 1090 data 221, 162,   2, 120, 189, 249, 159, 141
MF 1100 data  12, 221, 169,   8,  44,  13, 221, 240
GJ 1110 data 251, 202,  16, 240,  88,  96,  32, 253
 NI 1120 data 174,  32, 138, 173,  76, 247, 183,
```

Listing 6: This program creates "rcv.obj", the same code generated by assembling Listing 4 with PAL.

```
ND 100 rem program generator for "rcv.obj"
KD 110 n$="rcv.obj": rem  name of program
GK 120 nd=64: sa=828: ch=8062
```

For lines 130-260, use the standard generator program from page 5

```
BG 1000 data 186, 142, 124,   3, 169, 127, 141,  13
DF 1010 data 220, 173,  14, 220,  41, 191, 141,  14
MP 1020 data 220,  32, 106,   3, 133, 253,  32, 106
EG 1030 data   3, 133, 254,  32, 106,   3, 160,   0
NA 1040 data 145, 253, 240, 237, 169, 129, 141,  13
CE 1050 data 220, 174, 124,   3, 154,  96,  32, 188
KH 1060 data 246, 165, 145,  16, 239, 173,  13, 220
EB 1070 data  41,   8, 240, 242, 173,  12, 220,  96
```

The Link Between C and Assembly

David Godshall
Elkhart, Indiana

. . .How would you like to be able to access a machine language routine the same way you would access a C function?

If you own "Power C" from Spinnaker, you own an excellent and powerful implementation of the C language. If you don't, but have been thinking about buying a copy, the possibilities opened by this article may be enough to push you over the edge. Even if you don't own and don't plan to buy Power C, the techniques mentioned in this article may apply to other compilers that compile in two separate stages (from source into object, then from object into executable) or even for other types of programs.

The Problem. . .

I was writing some graphics routines to be used from C programs. I decided that, while Power C is probably the fastest compiler for the C-64 ("A Comparison of Language Speeds", Volume 7, Issue 5), nothing can beat hand-coding a piece of code for speed. I promptly set about coding assembly language routines to clear the bitmap, plot points, and all the other nice things you like to do to graphics screens quickly. In looking for a good way to use these routines from my C programs, I came across the SYS function. I tried it. I gave up on it. The main trouble with the SYS function is that it is little better than the BASIC SYS command. It assumes the code is already in memory (if it isn't, a special disk access is needed to put it there). It does *not* provide very descriptive access to the routines. The one thing it has over BASIC's SYS command is that it can handle parameters – three bytes' worth. Some of the functions I wanted to access, however, required more than 24 bits' worth of information.

So, giving up on the SYS function, I looked around and about and inside out for a better way. And I found it! My search led me to the internals of the object files – those mysterious files that are the limbo between pure source code and pure executable machine language. I didn't have a lot of hope. I've looked in real spaghetti files before and was afraid the object files would turn out to be too complicated for me to figure out without the aid of extensive documentation and weeks of personal interviews with Power C author Brian Hilchie. My fears were groundless.

Starting My Quest

To begin with, I already sort of knew what the linker does. It takes an object file, moves it to a specific address in memory, and then checks to see what the file needs to be linked with. It then gets those files, puts them at unique addresses, and checks what files they need. Finally, when all the files are linked in with everything they need and the linker can't find anything else to add, it writes the executable file containing all the object files all linked up nicely. But what is in an object file? How does it let the linker know what it wants and what it has to offer?

My first clue to the general format of the object files came from the mysterious and totally undocumented (in my manual, at least) LIB.C file. This is a nice utility that puts a list of all the global identifiers from multiple object files into one file so that the linker can find their functions and variables very easily. The first four files on the library side of the disk (STDLIB.L, STDLIB2.L, SYSLIB.L and SYSLIB2.L) can be examined and modified with this utility.

By looking at how LIB.C scans object files, I was able to determine that object files are divided into four sections. I call them the Code section, the Relocate section, the Global section, and the External section. Each section begins with a word (two bytes in low/high order) indicating how long the section is in bytes (for the Code section) or entries (for the other sections). The general format is summarized in Table 1.

Section 1: The Code Section

The first section looks familiar when viewed through a machine language monitor. It is straight machine language. Well, almost straight. There are a few differences that will be straightened out by the linker.

To begin with, there is some information that isn't known at the time the code is created. Any instruction referencing external functions or variables is going to have to have its operand filled in by the linker, so it doesn't matter what value is in the operand.

Secondly, one of the jobs of the linker is to decide where in memory to place the code. In order to enable the linker to do that easily, the code is generated by the compiler as if it were assembled to location $0000. In other words, if somewhere in the code you had a JMP instruction to transfer execution to the first instruction in the code section, it would be a JMP $0000 instruction. The linker can then relocate the code by calculating a new base address and adding it to the offsets contained in the operands of any instructions referencing a part of the code. But how does the linker know which instructions need to be adjusted for the new address and which are already pointing at the correct location (i.e. a ROM routine, a zero page location, etc.)? This is where the second section comes in:

Section 2: The Relocate Section

This section points out to the linker which instructions need to be adjusted during the address relocation process. If, for example, the instruction JSR $0073 is flagged by this section and the linker decided to relocate the code to base address $1153, then the instruction will be changed to **JSR $11c6**. If the JSR were *not* flagged by this section, if would remain JSR $0073.

This section consists of a list of addresses (as offsets from $0000) of instructions that need their operands relocated. The length word indicates how many addresses (of two bytes each) are in this section.

Section 3: The Global Section

This section tells the linker what the module has to offer to other modules. Any functions or variables that may be used by external routines are flagged in this section.

The length word indicates how many entries are in this section. Each entry will contain a name or identifier, a byte flag, and an address.

The name will be the one or more characters by which this variable or routine can be accessed. Remember that C is case sensitive and that identifiers coming out of the C compiler will be truncated to 8 characters. Terminate the name with a NULL (chr$(0)).

The byte flag tells the linker whether the entry is referring to a location in the code section or an absolute location. If the byte is a one, then the linker will know that it is referencing the code section and will adjust the address when the code section is relocated. A zero tells the linker that the address does not need to be relocated (it may be pointing to a ROM routine or some other stable location).

Section 4: The External Section

This section is sort of the opposite of the global section. It tells the linker what external routines and variables are needed by this module. It contains entries similar to those in the global section. Each entry consists of the name of the routine or variable to link in, a word specifying how to link it in (offset), and the address of the instruction accessing the external entity.

The word following the name allows several possibilities for linking in the address of the external entity. First of all, it allows you to link in the address of the entity, the address plus one, the address plus two, etc. You can add up to 8191 bytes to the address. Secondly, you can decide to either link in the whole address (for absolute instructions such as LDA xxxx) or just the high or low byte of the address (for immediate instructions such as LDA #<xxxx or LDA #>xxxx).

The way to specify these options is to take the number of bytes you want to add to the base address and multiply by four (to shift it into the upper 14 bits). Then you add 0, 1, or 2 depending on whether you want the whole address, the high byte, or the low byte respectively. The resulting value would go in this offset word.

Finishing up

To finish up the object file, just terminate it with two NULLs. Now you can give it the linker test! Beware, because the linker was created to link together modules created by the C compiler. Since the linker knows what type of object files the compiler is capable of creating, it isn't very error tolerant and will lock up on just about any irregularity. If you say there are five global entries, make sure you include exactly five. Make sure you terminate all identifiers with NULLs and the file with two NULLs. Et cetera.

Special Routines

There are several special external routines you may need to use when writing code to be linked in to work under the C environment. First is the **c$start** routine. This routine is included in every C program and is responsible for setting up the C environment. It does some setup work, calls the **main()** function, then does some clean-up work before returning control to the shell or BASIC. Thus, **c$start** must be the first thing to be called. The first instruction of the first file to be linked in must call this routine. But how do you know which of several files will be linked in first? To solve this, the C compiler puts a JMP c$start instruction as the first instruction in every module it generates. If there is a chance that your module might be linked in first, you would also want to put in a jump to the **c$start** external routine as the first instruction in your module.

Another important routine you will want to use is the **c$funct_init** routine. This is a routine that would be called first thing in any function you create. Normally, C functions call the routines **c$105** on entry and **c$106** just before returning instead of **c$funct_init**. **c$105** copies the local variables (locations $2b-4a) and parameters (cassette buffer $033c-$03fb) out of the way so the space can be used for new variables and parameters; **c$106** copies them in again upon completion of the function. These require a lot of overhead, so the **c$funct_init** routine comes in handy for small routines that will *not* need to use the local variable area (they can use the temporary locations $22-$2a and $4b-$60) and that will *not* call other routines that will use the variable area or the parameter area.

Unfortunately, to explain **c$105** and **c$106** in more detail would take us out of the scope of this article and into memory management.

Parameter Passing

One of the advantages of linking machine language routines through the object file as opposed to the SYS function is the ability to pass dozens of parameters. On the originating end, the values of the variables you are passing (or their addresses, if you are passing pointers) are stored in memory starting at $033c and in the same order as they were declared in the function descriptor. The accumulator is then set to reflect the number of bytes used up by the parameters, and the new function is called; it can then access the parameters directly from this memory. As an example, the function **FRED (Age, Name, Weight, Height)**; where **Age** is a character type, **Name** is a pointer to an array of characters, **Weight** is a floating point number, and **Height** is an integer; would store:

$033c – Age (one byte)
$033d – Name (low byte of pointer)
$033e – Name (high byte of pointer)
$033f – Weight
$0340 – Weight
$0341 – Weight (FP representation)
$0342 – Weight
$0343 – Weight
$0344 – Height (low byte)
$0345 – Height (high byte)

The accumulator would be set to 10. The called routine would naturally have to know what order the parameters are in and what type of variable each parameter is. If the called function needs to return a value, it should put it back into the cassette buffer at

location $033c. Since the value is not written until just before returning, you don't have to worry about overwriting what is already there.

An Example Is Worth Two Thousand Bytes

In order to clear up any questions you may still have, I will present a practical example of creating an object file from an assembly language file. While I used PAL as my assembler, you should be able figure out how to get your assembler to do some of the unusual things necessary to create an object file. Unfortunately, the current version of SYMASS, the PAL-compatible assembler, will not be able to assemble my example because it requires assembling to disk (since the code is assembled to location $0000).

Listing 2 is a Doodle program written in C. It requires four external routines, which it will get from Listing 1, the assembly language portion. You will have to compile the C portion, assemble the assembly language portion, and then link them together with the linker. You will then have an executable program that will let you draw on the hires screen with the IJKM diamond. The +, –, and / keys set the drawing mode to on, off, and flip respectively. RUN/STOP restores the normal text screen and exits the program. The program doesn't do any boundary checking – so don't try to draw off the screen or you may destroy something vital!

Listing 1 provides four functions: **Clear**, **Plot**, **FastKeys**, and **SlowKeys**. **Clear** fills any block of memory of any size with any byte. **Plot** allows you to manipulate any pixel on the graphic screen. **FastKeys** sets up an interrupt routine to speed up the keyboard repeat, and **SlowKeys** turns it off again.

Line 5 in listing 1 opens the object file to which it will write the object file. I am following a convention (which I suspect the author of Power C followed) of suffixing object files created from C source with a **.O** and object files created from Assembly source with **.OBJ**.

Lines 10 and 20 "fix" PAL so it writes the object file correctly for our purposes. Normally a machine language file begins with the object code origin address so that the kernal LOAD routine knows where to place the routine when you load it. The linker does not require that address and, in fact, gets confused by it. The pokes in line 20 replace the two JMP $FFD2 instructions that write the address to the file with do-nothing BIT $FFD2 instructions. If you have another assembler you will have to find a way to get around this problem. You may have to write a little program to strip the first two bytes off the object file after creating it.

Line 30 invokes PAL, and line 40 tells it to assemble to the file opened in line 10. In line 50 I tell the assembler to start assembling to location $0000 (minus two for the length word). I then define the filler label **xxxxxx** in line 60. I use this label in references to external entities since the assembler requires something. The linker will fill in the correct address. Line 120 sets up the jump to the setup routine in case this object file is the first one to be linked in.

Lines 100, 7010, 8020, and 9020 set up the length word for each of the sections. In line 100 it is just a matter of putting the end of the code section since the code starts at $0000. The length in line 7010 is calculated by taking the number of bytes defined in the relocate section, dividing by two (since the length is expressed in words instead of bytes), and subtracting one (to skip the length word). Calculating the length in the global and external sections is a little different. Here I use a label as if it were a variable, adding one for each entry, using PAL's left-arrow temporary assignment operator. Since calculating labels happens in the first pass and the code is written the second pass, it doesn't matter that the lines that increment the label (lines 8040, 8090, 8140, etc.) appear after the line putting the word in the file (line 8020 or 9020).

The **Clear** routine is in lines 160-390 and the global entry at lines 8090-8120 open this routine to allow access by other functions. Likewise, **Plot** in lines 500-1010 is opened by lines 8140-8170 as are the **FastKeys** and **SlowKeys** routines by the entries at lines 8190-8220 and 8240-8270 respectively.

Notice the global entry at lines 8290-8320 and the two external entries at lines 9280-9360 for the **irq%%** routine. Sometimes you may need to access a local routine or variable in a more specialized way than just by absolute addressing. Lines 1120 and 1140 need to access the local routine **irqkeys** by immediate addressing. The relocate section, however, only relocates absolute addressing instructions. In order to get it to work I had to treat the **irqkeys** routine as an external routine. This shows that local routines can be treated as external routines if necessary. Also, I chose to add two % symbols to the name to ensure that it doesn't interfere if you happen to define another routine named **irq** somewhere else.

In line 970 I am storing a value back into location $033c. This is to provide a return value so that the calling routine can check the new state of the pixel after the **Plot** routine is called.

In Conclusion. . .

I would like to thank Brian Hilchie for a powerful compiler that has raised the productivity value of the Commodore 64 by several notches. Thanks also for an elegant and straightforward object format. But why didn't he include this information in the documentation – to allow someone to make some money writing articles about it? I would suggest to Brian that, given the nature of C, machine language, and his specific implementation, it should not have been hard for him to include a #ASM and #ENDM set of compiler directives to allow inline assembly language. This would have made an attractive compiler virtually irresistible. I would recommend him adding it to a future update. After all, compared to writing a compiler, adding a simple assembler should be peanuts. He may be able to use PAL or SYMASS as a skeleton. If anyone could give me Brian Hilchie's address, I would like to be able to write to him myself.

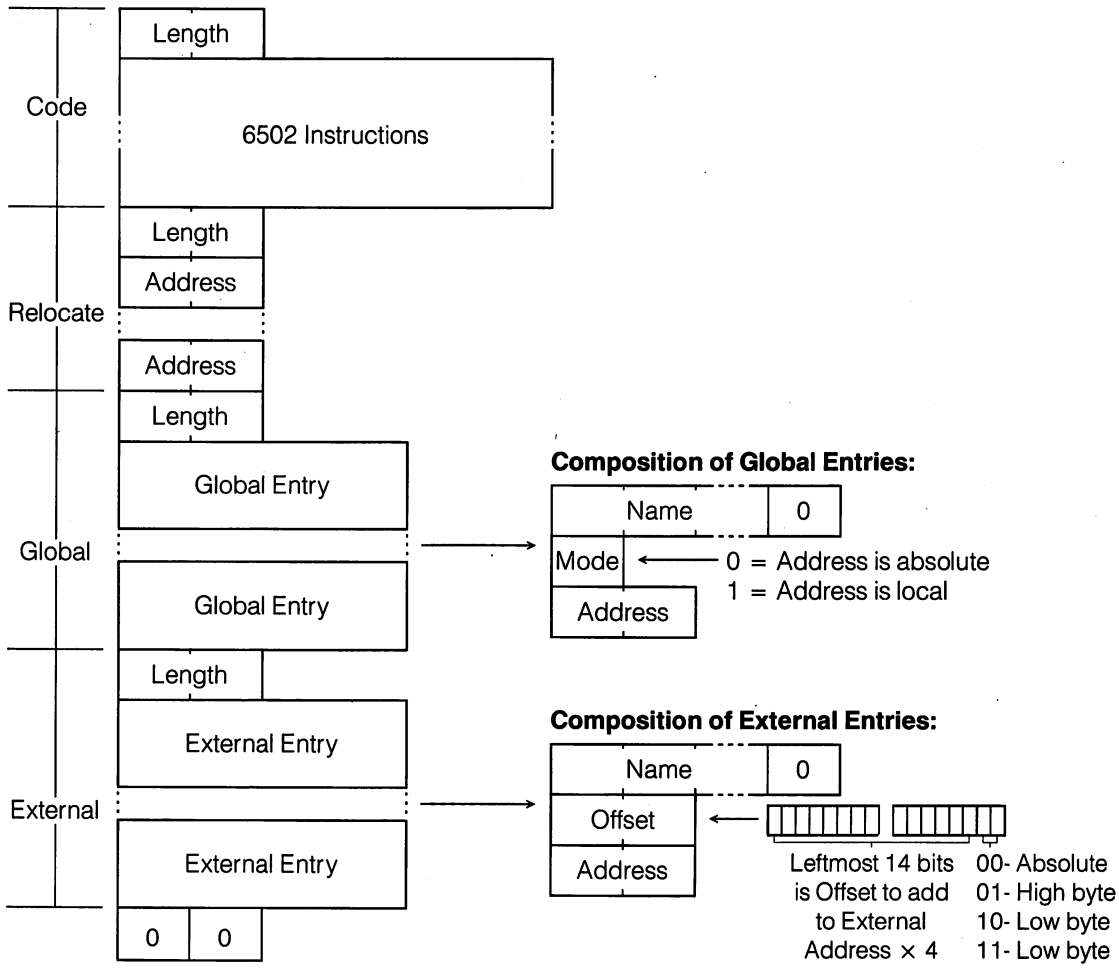
Those of you who want to take these ideas farther might want to tackle writing an assembler that would assemble source into object files of the type linkable by the C linker. You would probably need to add some pseudo ops like **.GLOB**, **.EXTN**, and **.FUNC**.

If you want to discuss specifics for such an assembler, or have any questions, problems, corrections or criticisms, I would love to hear from you. I can be reached at the following address:

David Godshall
137 Wagner
Elkhart, IN 46516

Fido-Mail or Net-Mail can be sent to me at node 11/205 – <G>oshen <T>owne <C>rier.

Table 1: Composition of Object Files



Listing 1: GRPLOT.PAL

```
HC  5   open 2,8,2,"@0:grplot.obj,s,w"
FM  10 pal=peek(701)+256*peek(702)
DB  20 poke pal+1759,44:poke pal+1764,44
PO  30 sys 700
JD  40 .opt o2
OH  50 *=-2
MF  60 xxxxxx = 0
PM  89 ;
OI  97 ;---------------
CM  98 ; code section
AJ  99 ;---------------
MA  100          .word creloc
EO  110 ;
FB  120 cstart   jmp  xxxxxx
IP  130 ;
KD  131 ;"function:
DO  132 ;"Clear (Address,Length,Byte)
NH  133 ;"unsigned int Address;
KA  134 ;"unsigned int Length;
DP  135 ;"char Byte;
EG  136 ;"
FP  137 ;"global:
CI  138 ;"  unsigned int Address;
BA  139 ;
FD  140 address  .word $e000
MA  150 ;
JM  160 clear    =*
NN  165 extn2    jsr   xxxxxx
DJ  170          lda   $033c      ;<addr
CB  180 rloc1    sta   !address
CI  190          sta   $22
MK  200          lda   $033d      ;>addr
JB  210 rloc2    sta   !address+1
DK  220          sta   $23
JO  230          lda   $0340      ;byte
DH  240          ldy   #$00
JE  250          ldx   $033f      ;>length
GF  260          beq   floop2
DJ  270 floop1   sta   ($22),y    ;fill a
CN  280          dey              ;page
NF  290          bne   floop1
KM  300          inc   $23        ;fill many
CJ  310          dex              ;pages
LH  320          bne   floop1
FJ  330          ldy   #0
JL  340 floop2   cpy   $033e      ;<length
FE  350          beq   fexit
IF  360          sta   ($22),y    ;fill part
ML  370          iny              ;of a page
JL  380          bne   floop2
LM  390 fexit    rts
FA  399 ;
NF  400 grrows   .word    0, 320, 640, 960,1280
AM  410          .word 1600,1920,2240,2560,2880
EL  420          .word 3200,3520,3840,4160,4480
BN  430          .word 4800,5120,5440,5760,6080
```

```
AA  440             .word 6400,6720,7040,7360,7680
HD  449 ;
AP  450 orbits      .byte 128,64,32,16,8,4,2,1
MI  460 andbits     .byte 127,191,223,239
FL  470             .byte 247,251,253,254
AG  490 ;
AI  491 ; function:
JL  492 ;*  char Plot (x,y)
NE  493 ;*  unsigned int x,y;
EG  494 ;
FG  495 ;
AH  500 plot        =*
DD  505 extn3       jsr   xxxxxx
MJ  510             lda   $033e       ;y coord
CC  520             lsr   a
MC  530             lsr   a
IE  540             and   #254
FP  550             tay
LD  560 rloc3       lda   grrows,y    ;get row
AG  570             clc               ;and add
FI  580 rloc4       adc   !address    ;bitmap
PL  590             sta   $22         ;address
MP  600 rloc5       lda   grrows+1,y
ND  610 rloc6       adc   !address+1
DD  620             sta   $23
OI  630             lda   $033c       ;x coord lo
GK  640             and   #%11111000
CA  650             adc   $22
IF  660             sta   $22
BK  670             lda   $033d       ;x coord hi
DC  680             adc   $23
JH  690             sta   $23
KF  700             lda   $033e       ;y coord
KO  710             and   #%00000111
PJ  720             tay
MP  740             lda   $033c       ;x coord lo
CB  750             and   #%00000111
CM  760             sta   $24
LJ  770             sei
ID  780             lda   $01         ;swap all
HP  790             pha               ;rom/io out
ME  800             lda   #$30
HO  810             sta   $01
LN  820             lda   ($22),y     ;check
NL  830 extn1       ldx   !xxxxxx     ;plot type
AJ  840             beq   bitoff      ;and modify
GJ  850             cpx   #1          ;pixel
LO  860             beq   biton
OE  870 bitflip     ldx   $24         ;invert
NL  880 rloc7       eor   !orbits,x
DB  890 rloc8       jmp   pexit
BA  900 biton       ldx   $24         ;pixel on
EG  910 rloc9       ora   !orbits,x
AJ  920 rloc10      jmp   pexit
LJ  930 bitoff      ldx   $24         ;pixel off
LK  940 rloc11      and   !andbits,x
JI  950 pexit       sta   ($22),y     ;replace
BI  960 rloc12      and   !orbits,x   ;byte and
ML  970             sta   $033c       ;return
FP  980             pla               ;bit state.
DB  990             sta   $01         ;restore
FD 1000               cli               ;io/roms.
ON 1010               rts
IL 1090 ;
IN 1091 ; function:
AA 1092 ;*  FastKeys ()
LL 1093 ;
OA 1100 fastkey     =*
PO 1110             sei
IJ 1120 extn4       lda   #<irqkeys
FC 1130             sta   $0314
IK 1140 extn5       lda   #>irqkeys
KD 1150             sta   $0315
GB 1160             cli
OH 1170             rts
JI 1299 ;
GB 1300 irqkeys     =*
FE 1310             lda   #$01
KA 1320             sta   $028b
FF 1330             lda   #$00
PB 1340             sta   $028c
PG 1350             jmp   $ea31
EO 1390 ;
EA 1391 ; function:
MG 1392 ;*  SlowKeys ()
HO 1393 ;
IG 1400 slowkey     =*
LB 1410             sei
HD 1420             lda   #<$ea31
BF 1430             sta   $0314
HE 1440             lda   #>$ea31
GG 1450             sta   $0315
CE 1460             cli
KK 1470             rts
OD 1480 ;
AD 6997 ;-------------------
CI 6998 ; relocate section
CD 6999 ;-------------------
PA 7000 creloc      =*
ED 7010             .word (cglobal-creloc)>1-1
CO 7020 ;
EE 7030             .word rloc1
AF 7040             .word rloc2
DO 7050             .word rloc3  ;the addrs
BD 7060             .word rloc4  ;of all
DL 7070             .word rloc5  ;instructions
GG 7080             .word rloc6  ;accessing
FD 7090             .word rloc7  ;local
II 7100             .word rloc8  ;variables.
EK 7110             .word rloc9
OC 7120             .word rloc10
LD 7130             .word rloc11
IE 7140             .word rloc12
CL 7996 ;
LG 7997 ;-----------------
AO 7998 ; global section
NG 7999 ;-----------------
LJ 8000 cglobal     =*
PP 8010 numglob =     0
MF 8020             .word numglob
EN 8030 ;
JB 8040 numglob _ numglob+1
FO 8050             .asc "Address":.byt 0
IL 8060             .byt 1
KG 8070             .word address
GA 8080 ;
LE 8090 numglob _ numglob+1
EL 8100             .asc "Clear":.byt 0
KO 8110             .byt 1
AI 8120             .word clear
ID 8130 ;
NH 8140 numglob _ numglob+1
KG 8150             .asc "Plot":.byt 0
MB 8160             .byt 1
DG 8170             .word plot
KG 8180 ;
```

```
PK  8190 numglob _ numglob + 1
ED  8200         .asc "FastKeys":.byt 0
OE  8210         .byt 1
JD  8220         .word fastkey
MJ  8230 ;
BO  8240 numglob _ numglob + 1
JI  8250         .asc "SlowKeys":.byt 0
AI  8260         .byt 1
JJ  8270         .word slowkey
OM  8280 ;
DB  8290 numglob _ numglob + 1
FJ  8300         .asc "irq%%":.byt 0
CL  8310         .byt 1
NN  8320         .word irqkeys
AA  8330 ;
KJ  8996 ;
AA  8997 ;------------------
HM  8998 ; external section
CA  8999 ;------------------
DP  9000 cextern  = *
KO  9010 numext   =   0
BG  9020         .word numext
ML  9030 ;
BE  9040 numext  _ numext + 1
BE  9050         .asc "c$start":.byt 0
JC  9060         .word 0
NI  9080         .word cstart
IP  9090 ;
NH  9100 numext  _ numext + 1
EL  9110         .asc "PlotType":.byt 0
FG  9120         .word 0
LK  9140         .word extn1
ED  9150 ;
JL  9160 numext  _ numext + 1
CO  9170         .asc "c$funct[]init":.byt 0
BK  9180         .word 0
JO  9200         .word extn2
AH  9210 ;
FP  9220 numext  _ numext + 1
OB  9230         .asc "c$funct[]init":.byt 0
NN  9240         .word 0
HC  9260         .word extn3
MK  9270 ;
BD  9280 numext  _ numext + 1
DH  9290         .asc "irq%%":.byt 0
NB  9300         .word 2
LF  9310         .word extn4
ON  9320 ;
DG  9330 numext  _ numext + 1
FK  9340         .asc "irq%%":.byt 0
NE  9350         .word 1
PI  9360         .word extn5
EI  9998 ;
DB  9999         .word 0              ;done!
```

Listing 2: DOODLE.C

```
/*
    doodle.c

    by David Godshall
*/

char PlotType;

main ()
{
    char *Pointer, Key, Store1, Store2, Store3;
    unsigned int Loop, X, Y;

    highmem (0xCC00);

    Pointer = 0xDD00;
    *Pointer = (Store1 = *Pointer) & 252;
    Pointer = 0xD011;                    /* Turn on Graphics          */
    *Pointer = (Store2 = *Pointer) | 32;
    Pointer = 0xD018;
    Store3 = *Pointer;
    *Pointer = 0x38;

    Pointer = 0x028a;                    /* Turn on key repeat        */
    *Pointer = 128;
    FastKeys ();

    Clear (0xCC00, 1000, 93);            /* Clear colour screen       */
    Clear (0xE000, 8000, 0);             /* Clear bitmap              */
    PlotType = 1;
    X = 160;
    Y = 100;
    Plot (X,Y);
    while ((Key = waitkey()) != 3)
    {
        switch (Key)
        {
            case 'i' :
            case 'I' : Plot (X,--Y);     /* Allow user to draw lines  */
                       break;            /* by using the I, J, K, M   */
            case 'm' :                   /* diamond. -, +, and / set  */
            case 'M' : Plot (X, ++Y);    /* clear, set, or flip mode  */
                       break;            /* respectively.  STOP exits */
            case 'j' :
            case 'J' : Plot (--X,Y);
                       break;
            case 'k' :
            case 'K' : Plot (++X,Y);
                       break;
            case '-' : PlotType = 0;
                       Plot (X, Y);
                       break;
            case '+' : PlotType = 1;
                       Plot (X, Y);
                       break;
            case '/' : PlotType = 2;
                       Plot (X, Y);
        }
    }

    SlowKeys ();

    Pointer = 0xDD00;
    *Pointer = Store1;
    Pointer = 0xD011;                    /* Restore Text mode         */
    *Pointer = Store2;
    Pointer = 0xD018;
    *Pointer = Store3;
}


#define GETIN  0xFFE4

char a, x, y,
     *numkeys = 198;

/* Waits for user to press a key */

int waitkey ()

{
    while (*numkeys == 0)
        ;
    sys (GETIN, &a, &x, &y);
    return a;
}
```

Maintaining the POWER C Library

Eric Giguere
Waterloo, ON

. . .It would be easier, I thought, if I could somehow include my own routines into the standard library. . .

The Power C compiler for the Commodore 64 is one of the best software investments I have ever made. Although the Commodore 64 is not the best development tool for programming in C, Brian Hilchie's compiler functions better than I expected any C compiler to do. As a result, I have been constantly using it over the past few months, slowly building up a library of routines for my personal use. Unfortunately, this was a problem in itself: I really hated having to manually link in my own library routines. It would be easier, I thought, if I could somehow include my own routines into the standard library (built-in functions) provided with the compiler. After a bit of research, I realized how ridiculously easy this would be, and proceeded to write up a short library maintenance program.

Library Structure

Included on the system disk that comes with the compiler are three library files. Each library file is a simple sequential file consisting of sets of two ASCII strings separated by null characters (ASCII 0). Each set consists of a function name (significant to eight characters) and the filename of the object file that function is to be found in. The linker uses these files to link the proper object modules with your own programs. The three libraries (all suffixed with a ".l" on the disk) are:

syslib routines for the compiler's own internal use
stdlib the standard library as described in the Power C manual
math math routines

I don't recommend altering the first library in any way, but the other two are fair game. Since the structure of a library file is so simple, adding a function to a library is really a matter of appending two strings to the end of the library file. Program 1, the Updater, does just that. It adds a function to the specified library file. If, for example, you wish to add the function "poke", found in "poke.obj", to the library file "stdlib", you would simply type in the library name, the function name, and the filename. The program automatically adds the required ".l" and ".obj" suffixes to the library name and the filename and proceeds to add the function to the library.

Creating a New Library

The Updater program was an easy way for adding functions to a library file, but I soon realized that it would be useful to make other changes as well. Many of the files that are included on the library disk are not used often enough in my programs to warrant the space they take up on the library disk. I decided that it would be useful to take the stdlib file and remake it to best suit my own needs. Program 2, the Library Editor, is another simple program to do just that.

When creating your own library you have two choices: modify the existing stdlib file or create your own library file. Creating a new stdlib file is very simple. First, copy the old stdlib file onto a fresh diskette. You might also want to copy the linker program itself onto that diskette. Then load up the Library Editor program. Use option L to load in the stdlib file. Then use the view option to examine the file. Choose which functions you wish to delete and use the D option to delete them (one at a time, by function name). When done, use the A option to add your own functions. Then save the file back onto the diskette with the S option. To complete the new library, quit the editor and copy all the necessary object files onto the new library diskette. You might want to add another option to the library editor to print out a list of all the filenames required just to keep track of things. To use the new library, just make sure your new library disk is in the drive when you link in the library by pressing the up-arrow key. It's as simple as that. (By the way, don't forget to copy the syslib file and its files over to the new library disk as well.)

Making your own library file is just as simple. First, do the following:

```
open 1,8,5,"0:mylib.l,s,w"
close1
```

(Of course, you can use any name you want instead of "mylib" – just don't omit the ".l".) This creates an empty library file with the specified name. Then use the updater program to add the required information to the file. When you are linking, simply type the library name (again, don't forget the ".l" suffix) and the needed files from that library, if any, will automatically be linked. Then continue with the linking process as usual.

Final Notes

Keeping your own personal library helps to streamline the compilation process, and thus saves both time and frustration. While this article was concerned primarily with the Power C compiler, the general methods I used can also be applied to other compilers and machines, although their library schemes may be a lot different.

Listing 1: Library Editor

```
CJ  10 rem c-power library editor
CD  20 rem
AP  30 rem by eric giguere
GE  40 rem
NH  50 poke 53280, 14: poke 53281, 1
EN  60 max = 200: dim fc$(max), fl$(max)
CB  70 nu$ = chr$(0)
OA  80 open 15,8,15
GN  100 print chr$(14)"[clr][blue]C Library Maintenance
        -- by Eric Giguere";
AK  110 print
LF  120 print "[down] [black][A]dd a function"
KO  130 print "[down] [D]elete a function"
LJ  140 print "[down] [L]oad a library"
KJ  150 print "[down] [S]ave the library"
LD  160 print "[down] [Q]uit"
HO  170 print "[down] [V]iew the library"
MN  180 print "[down] Please select: ";: gosub 1000
        : print c$: if c$ = "q" then end
FO  190 if c$<>"l" then 350
LL  200 print "[clr][down] Library name: ";: gosub 1010
CH  210 lib$ = left$(in$, 12) + ".l": n = 0: a$ = ""
DN  220 open 2,8,5, "0:" + lib$ + ",s,r"
AG  230 gosub 1500: if e then 1550
JC  240 print "[down] loading ";lib$;". . ."
MB  250 get#2,b$: if b$<>"" and st = 0 then a$ = a$ + b$
        : goto 250
HM  260 if st<>0 then 300
LD  270 n = n + 1: fc$(n) = a$: a$ = ""
ND  280 get#2,b$: if b$<>"" and st = 0 then a$ = a$ + b$
        : goto 280
HN  290 fl$(n) = a$: a$ = "": if st = 0 then 250
FC  300 close 2: goto 100
EK  350 if c$<>"s" then 450
BD  360 if n = 0 then 100
JD  370 print "[clr][down] Are you sure? (y/n) ";
        : gosub 1000: if c$<>"y" then 100
NC  380 print c$: print "[down] Saving ";lib$;". . ."
LC  390 print#15,"s0:" + lib$
KE  400 open 2,8,5,"0:" + lib$ + ",s,w": gosub 1500
        : if e then 1550
CK  410 for i = 1 to n
DI  420 print#2,fc$(i);nu$;fl$(i);nu$;
FH  430 next: close 2: goto 100
HB  450 if c$<>"v" then 550
FJ  460 if n = 0 then 100
GN  470 print "[clr][down] Function[8 spces]Filename"
CB  480 print
CP  490 for i = 1 to n
DB  500 print " ";fc$(i);tab(17);fl$(i)
IN  510 wait 197,64: next
OF  520 print "[down] Press a key. . . ";: gosub 1000
        : goto 100
KC  550 if c$<>"a" then 650
JP  560 if n = 0 then 100
FC  570 print "[clr][down] Add a function"
GO  580 print "[down] Function name: ";: gosub 1010
FF  590 n = n + 1: fc$(n) = left$(in$, 8)
JN  600 print "[down] Filename: ";: gosub 1010
PD  610 fl$(n) = left$(in$, 12) + ".obj"
BA  620 goto 100
HH  650 if c$<>"d" then 100
IF  655 if n = 0 then 100
DO  660 print "[clr][down] Delete a function"
AE  670 print "[down] Function name: ";: gosub 1010
IN  680 fc$ = left$(in$, 8): j = 0
LF  690 for i = 1 to n: if fc$ = fc$(i) then j = i: i = n + 1
FJ  700 next: if j = 0 then 100
KC  710 if j = n then n = n-1: goto 100
IK  720 for i = j + 1 to n: fc$(i-1) = fc$(i): fl$(i-1) = fl$(i)
        : next: n = n-1
PG  730 goto 100
HO  999 end
GF  1000 poke 198,0: poke 204, 0: wait 197,64,64
        : get c$: poke 204, 1: return
LE  1010 open 1,0: input#1, in$: close1: print: return
CG  1500 input#15,e,e$: if e<20 then e = 0
CA  1510 return
PB  1550 print "[clr][down] Disk error -- #";e
JG  1560 print "[17 spcs]";e$
JO  1570 close 2: close 15: end
```

Listing 2: Updater

```
AE  10 rem c-power library updater
CD  20 rem
AP  30 rem by eric giguere
GE  40 rem
KL  50 print "[clr][down]Insert your library disk into
        drive 0"
FH  60 print "[down] Library name: ";: gosub 1000
        : lib$ = left$(in$, 14) + ".l"
BE  70 print "[down] Function name: ";: gosub 1000
        : fc$ = left$(in$, 8)
NH  80 print "[down] Filename: ";: gosub 1000
        : fl$ = left$(in$, 12) + ".obj"
GG  90 print "[down] Updating ";lib$;". . ."
IC  100 open 2,8,5,"0:" + lib$ + ",s,a"
OI  110 print#2,fc$;chr$(0);
KK  120 print#2,fl$;chr$(0);
IF  130 close 2
JA  140 print "[down] Done. Another function? (y/n) ";
        : gosub 1000
EG  150 if left$(in$, 1) = "y" then 70
AK  160 end
BE  1000 open 1,0: input#1,in$: close 1: print: return
```

A Better Syntax For Kernal Device I/O

Keath Milligan
Austin, Texas

. . .Countless times I have watched my second drive collect dust because programs simply wouldn't talk to it.

Compared to other operating systems in its class, the C-64's "Kernal" offers quite a bit of I/O power to the user. However, it does have quite a few shortcomings, one of them being the way it references the different devices.

The Problem

Most operating systems expect the device number (and sometimes other information, depending on the system) to be part of the filename. For instance, a PC user might use:

```
LOAD "B:BEEPBOOP.BAS"
```

to load up his favourite music program. Note that there is no ",8" or anything equivalent. But take a closer look at that filename. The device (in this case, disk drive B) is specified right there with the filename. The Amiga uses the same method as the PC, but with different device names.

This method gives users of these systems quite a bit of choice when they are asked for a filename. For instance, if a program running on a PC asks for a filename, the user can usually respond with any valid device name or filename, thus giving him the choice of any of the devices connected to his system.

The 64 user, on the other hand, could only give the filename, and the program would probably default to drive 8. Countless times I have watched my second drive collect dust because programs simply wouldn't talk to it.

Actually, Commodore DOS does have this sort of filename syntax for specifying drives, but it applies to only the drives in a particular disk unit. If you have a single drive unit, like the 1541, this DOS feature is mostly useless, so the unfortunate 64 user is out of luck again.

The Program

Obviously, the more standard method of "devicename:filename" is superior to the syntax used on Commodore's 8-bit machines. It gives the user access to all the devices connected to the computer *and* does not require extra programming effort.

Luckily for us, Commodore did have a few good ideas, and one of their best was the use of page 3 vectors. By intercepting some of these vectors it's possible to change the way the Kernal communicates with devices. The program accompanying this article uses this method to give the 64 a variation of the same sort of device/file name syntax used by MS-DOS and Amiga-DOS.

To install the new filename parser, run the BASIC loader. The parser can be located at any even page boundary, i.e. any address that is evenly divisible by 256. Note that the loader doesn't change any of the BASIC memory pointers, so if you wish to locate the code at an address in the main memory area (2048-40959), you'll have to adjust the pointers accordingly to prevent BASIC from overwriting the code. The enhanced syntax can be disabled by simply pressing RUN-STOP/RESTORE.

Now you can reference devices by using the general syntax of 'D', 'P' or '#' followed by the device number followed by a colon in front of the filename. For example:

```
"D9:FILENAME"
```

specifies a file on device 9. ('D', 'P' and '#' can be used interchangeably)

Secondary addresses can now be specified from within the filename as well. To do this, follow the device number with a comma followed by the secondary address. For example: "#8,2:FILENAME". This can most useful for printer files.

The filename is optional, so the printer and other devices can be referenced easily. Examples: "P4,7:", "#0:".

Note the colons, which must be present even though the filenames are missing. You should also be aware that device numbers and secondary addresses specified in this manner will take precedence over those specified outside the filename. Think of the device numbers and secondary addresses specified within the filename as adjustments to those specified outside the filename. For instance,

```
LOAD"D8:FILENAME",0,1
```

is equivalent to

```
LOAD"FILENAME",8,1
```

and

```
OPEN 1,8,8,"P4,7:"
```

is equivalent to

```
OPEN 1,4,7
```

The 'D' or 'P' may be specified in upper or lower case. If the parser encounters anything that it doesn't understand, it simply passes the entire filename untouched to the Kernal. This is to ensure that the parser will not interfere with disk commands and the like.

With the enhancement active, you must specify any reference to the cassette drive within the filename:

```
LOAD"#1:FILENAME"
```

The enhancement defaults to device number 8. Therefore LOAD"FILENAME" will look for the file on device number 8.

Here are some examples of the enhanced syntax in use:

```
LOAD"D9:FILENAME"

LOAD"FILENAME"  (loads from device 8)

SAVE"D1:FILENAME"

OPEN1,0,0,"P4,27:"

LOAD"D8,1:*"  (same as LOAD"*",8,1)
```

The enhancement will also work with BASIC and some machine language programs so long as there are no vector or memory conflicts. If a BASIC program prompts you for a filename, you can, in most cases, use the enhanced syntax. This will allow you to access your choice of devices. You may have to put a quote (") before the filename because the INPUT statement in BASIC can't handle colons or commas. Try it without the quote first – some programs have their own improved input routines.

This enhanced parser might work well in a module for a TransBASIC dialect. Commands such as DIR or BLOAD could be written that would no longer have to look for device numbers.

Listing 1: BASIC Loader for New File Name Parser

```
JB  100 print"start address  49152"
AP  110 printchr$(145);tab(13);
ON  120 inputsa:a=sa
PB  130 ms=int(a/256):ls=a-ms*256
AA  140 ifls<>0thenprint"must be even page boundary"
        :end
NK  150 readv:ifv=-1then190
PO  160 ch=ch+v
MK  170 if(v=192)or(v=193)thenv=v-192:v=v+ms
EA  180 pokea,v:a=a+1:goto150
CJ  190 ifch<>43286thenprint"checksum error.":end
MM  200 print"enhanced filename syntax enabled."
LA  210 sys sa
OA  220 print
PI  230 print"to change default device use:  "
AL  240 print"     poke";sa+134;",device"
KP  250 end
IH  260 :
ME  270 data 169,  31, 160, 192, 141,  26,   3, 140
NE  280 data  27,   3, 169,  37, 160, 192, 141,  48
NE  290 data   3, 140,  49,   3, 169,  45, 160, 192
OD  300 data 141,  50,   3, 141,  51,   3,  96,  32
FI  310 data  51, 192,  76,  74, 243,  72,  32,  51
HP  320 data 192, 104,  76, 165, 244,  32,  51, 192
FO  330 data  76, 237, 245, 165, 187, 141,  21, 193
EK  340 data 165, 188, 141,  22, 193, 165, 186, 141
ON  350 data  39, 193, 165, 185, 141,  40, 193, 165
AP  360 data 183, 141,  42, 193,  32,  20, 193, 201
GJ  370 data  35, 240,   8, 201,  68, 240,   4, 201
OH  380 data  80, 208,  32, 162,   0, 141,  41, 193
EH  390 data  32,  20, 193, 141,  43, 193, 201,  58
II  400 data 240,  34, 201,  44, 240,  30, 201,  48
OI  410 data 144,   9, 201,  58, 176,   5, 232, 224
GJ  420 data   3, 208,  15, 165, 183, 240,  10, 165
BN  430 data 186, 201,   1, 208,   4, 169,   8, 133
IH  440 data 186,  96, 208, 212, 224,   0, 240, 235
FB  450 data 165, 187, 141,  21, 193, 165, 188, 141
GI  460 data  22, 193, 165, 183, 141,  42, 193,  32
CA  470 data  20, 193, 173,  21, 193, 133,   3, 173
GA  480 data  22, 193, 133,   4, 138,  32,  44, 193
FC  490 data 176,  48, 165,   5, 141,  39, 193,  32
KN  500 data  20, 193, 201,  58, 240,   6, 201,  44
FE  510 data 240,  35, 208, 243, 173,  21, 193, 133
PJ  520 data 187, 173,  22, 193, 133, 188, 173,  42
KF  530 data 193, 133, 183, 173,  40, 193, 133, 185
NP  540 data 173,  39, 193, 133, 186, 201,   1, 208
GF  550 data   1,  96,  76, 123, 192, 162,   0, 173
JG  560 data  21, 193, 133,   3, 173,  22, 193, 133
NG  570 data   4,  32,  20, 193, 201,  58, 240,  14
LC  580 data 232, 224,   4, 240,   8, 201,  48, 144
EG  590 data   4, 201,  58, 144, 236,  96, 138, 240
AE  600 data 252,  32,  44, 193, 165,   5, 141,  40
FC  610 data 193,  76, 196, 192, 173, 255, 255,  41
MK  620 data 127, 238,  21, 193, 208,   3, 238,  22
BM  630 data 193, 206,  42, 193, 201,   0,  96,   0
OI  640 data   0,   0,   0,   0, 141, 123, 193, 169
JH  650 data   0, 168, 174, 123, 193, 133,   5, 133
DA  660 data   6,  24,  32,  85, 193, 176,  21, 177
PG  670 data   3,  41,  15,  24, 101,   5, 133,   5
LM  680 data 144,   5, 230,   6,  56, 240,   5,  24
DP  690 data 200, 202, 208, 230,  96,   6,   5,  38
MO  700 data   6, 176,  28, 165,   6,  72, 165,   5
FK  710 data  72,   6,   5,  38,   6, 176,  17,   6
GO  720 data   5,  38,   6, 176,  11, 104, 101,   5
JD  730 data 133,   5, 104, 101,   6, 133,   6,  96
AC  740 data 104, 104,  96,   0,  -1
```

Listing 2: PAL Source for New File Name Parser

```
JL 1000 sys700
MC 1010 ;*********************************
JL 1020 ;*            file-spec parser           *
MA 1030 ;*       sep 21, 1987  version 1.0       *
KD 1040 ;*                                       *
EB 1050 ;*             keath milligan            *
NC 1060 ;*           11909 swan drive            *
BH 1070 ;*           austin, tx 78750            *
KK 1080 ;*           (512) 331-8451              *
MH 1090 ;*********************************
CM 1100 ;
MM 1110 ;
IM 1120 .opt oo
AO 1130 ;
MB 1140 fnlen    =   $b7
BP 1150 la       =   $b8
MA 1160 sa       =   $b9
EB 1170 fa       =   $ba
FC 1180 fnadr    =   $bb
NH 1190 aptr     =   $03
LI 1200 bptr     =   $05
AD 1210 ;
KO 1220          lda  #<openbp
AF 1230          ldy  #>openbp
DM 1240          sta  794
AD 1250          sty  795
OO 1260          lda  #<loadbp
EF 1270          ldy  #>loadbp
CO 1280          sta  816
PE 1290          sty  817
JB 1300          lda  #<savebp
PH 1310          ldy  #>savebp
AB 1320          sta  818
NB 1330          sta  819
IC 1340          rts
ML 1350 ;
MK 1360 openbp   jsr  parse
PG 1370          jmp  $f34a
KN 1380 ;
ND 1390 loadbp   pha
GK 1400          jsr  parse
KA 1410          pla
MM 1420          jmp  $f4a5
MA 1430 ;
GO 1440 savebp   jsr  parse
MA 1450          jmp  $f5ed
KC 1460 ;
PK 1470 ;*** parse routine ***
OD 1480 ;
OP 1490 parse    lda  fnadr
HB 1500          sta  fnptr+1
FJ 1510          lda  fnadr+1
NC 1520          sta  fnptr+2
KC 1530          lda  fa
NK 1540          sta  tdev
LE 1550          lda  sa
LH 1560          sta  tsa
FL 1570          lda  fnlen
DL 1580          sta  ctr
DH 1590          jsr  getfchr
FE 1600          cmp  #"#"
EC 1610          beq  checkfn
ML 1620          cmp  #"d"
ID 1630          beq  checkfn
EP 1640          cmp  #"p"
JK 1650          bne  chkdev
DA 1660 checkfn  ldx  #0
NM 1670          sta  tdname
LD 1680 getnext  jsr  getfchr
JE 1690          sta  tsep
OO 1700          cmp  #":"
JG 1710          beq  fnok
IN 1720          cmp  #","
NH 1730          beq  fnok
IP 1740          cmp  #"0"
EO 1750          bcc  chkdev
DD 1760          cmp  #"9"+1
ID 1770          bcs  chkdev
AN 1780          inx
NF 1790          cpx  #3
FN 1800          bne  gnext
NF 1810 chkdev   lda  fnlen
ML 1820          beq  parsex
GF 1830          lda  fa
CG 1840          cmp  #1
FM 1850          bne  parsex
HM 1860          lda  #8          ;default drive
ML 1870          sta  fa
KC 1880 parsex   rts
BE 1890 gnext    bne  getnext
AE 1900 fnok     cpx  #0          ;# of digits
CM 1910          beq  chkdev
DC 1920          lda  fnadr
FM 1930          sta  fnptr+1
DE 1940          lda  fnadr+1
LN 1950          sta  fnptr+2
LD 1960          lda  fnlen
JD 1970          sta  ctr
GF 1980 ;get pointer on device number
DA 1990          jsr  getfchr
NM 2000          lda  fnptr+1
FK 2010          sta  aptr
DO 2020          lda  fnptr+2
FM 2030          sta  aptr+1
MK 2040          txa
GM 2050          jsr  asc2wor
NK 2060          bcs  passit
PH 2070          lda  bptr        ;device number
JM 2080          sta  tdev
DF 2090 iloop    jsr  getfchr
NO 2100 parsec   cmp  #":"
ML 2110          beq  finishfs
IG 2120          cmp  #","
KC 2130          beq  chksa
ID 2140          bne  iloop
ED 2150 finishfs lda  fnptr+1
```

```
BF 2160          sta  fnadr
JH 2170          lda  fnptr+2
BH 2180          sta  fnadr+1
HN 2190          lda  ctr
JG 2200          sta  fnlen
HM 2210          lda  tsa
HC 2220          sta  sa
BC 2230          lda  tdev
OC 2240          sta  fa
MP 2250          cmp  #1
OF 2260          bne  passit
KM 2270          rts
AH 2280 passit   jmp  chkdev
DH 2290 chksa    ldx  #0
JP 2300          lda  fnptr+1
BN 2310          sta  aptr
PA 2320          lda  fnptr+2
BP 2330          sta  aptr+1
EB 2340 csaloop  jsr  getfchr
IH 2350          cmp  #":"
EL 2360          beq  csaout
OB 2370          inx
NK 2380          cpx  #4
DK 2390          beq  csaerr
MI 2400          cmp  #"0"
JH 2410          bcc  csaerr
HM 2420          cmp  #"9"+1
AJ 2430          bcc  csaloop
MC 2440 csaerr   rts
ED 2450 csaout   txa
JO 2460          beq  csaerr
KG 2470          jsr  asc2wor
OD 2480          lda  bptr
NB 2490          sta  tsa
GG 2500          jmp  finishfs
EE 2510 ;
KK 2520 getfchr  =    *
EF 2530 fnptr    lda  $ffff      ;dummy addr
MK 2540          and  #%01111111 ;mask high-bit
DB 2550          inc  fnptr+1
OB 2560          bne  gfc
JC 2570          inc  fnptr+2
LB 2580 gfc      dec  ctr
OE 2590          cmp  #0
EB 2600          rts
IK 2610 ;
EF 2620 tdev     .byt 0
IC 2630 tsa      .byt 0
OA 2640 tdname   .byt 0
BG 2650 ctr      .byt 0
DK 2660 tsep     .byt 0
EO 2670 ;
CL 2680 ;*********************************
PN 2690 ;*    convert decimal ascii to word    *
GK 2700 ;*          .a = length of string      *
JH 2710 ;*          aptr points to string      *
DG 2720 ;*        ret-bptr = 16 bit word       *
MK 2730 ;*          cs = error, cc = ok        *
OO 2740 ;*********************************
ED 2750 ;
OD 2760 ;
JE 2770 asc2wor  sta  declen
HM 2780          lda  #0
FL 2790          tay
HF 2800          ldx  declen
GM 2810          sta  bptr
NF 2820          sta  bptr+1
EI 2830          clc
AC 2840 asc2wor1 jsr  by10       ;multiply by 10
FC 2850          bcs  asc2worx   ;cs - overflow
KA 2860          lda  (aptr),y
JO 2870          and  #%00001111
GL 2880          clc
KM 2890          adc  bptr
AC 2900          sta  bptr
CO 2910          bcc  asc2wor2
DK 2920          inc  bptr+1
DP 2930          sec
LH 2940          beq  asc2worx   ;eq - overflow
EF 2950 asc2wor2 clc
AH 2960          iny
BF 2970          dex
NE 2980          bne  asc2wor1
FC 2990 asc2worx rts
OC 3000 ;
LH 3010 ; by10- multiply by 10
CE 3020 ;
LK 3030 by10     asl  bptr       ;x 2
EF 3040          rol  bptr+1
NN 3050          bcs  by10r      ;cs - overflow
PA 3060          lda  bptr+1
KH 3070          pha
GJ 3080          lda  bptr
OI 3090          pha
OG 3100          asl  bptr       ;x 2
KJ 3110          rol  bptr+1
JC 3120          bcs  by10x      ;cs - overflow
MI 3130          asl  bptr       ;x 2
IL 3140          rol  bptr+1
HE 3150          bcs  by10x      ;cs - overflow
AO 3160          pla
CO 3170          adc  bptr
ID 3180          sta  bptr
OP 3190          pla
NI 3200          adc  bptr+1
DO 3210          sta  bptr+1
EF 3220 by10r    rts
CD 3230 by10x    pla
AD 3240          pla
OJ 3250          rts
CD 3260 ;
GC 3270 declen   .byt 0
```

A RAM Expansion Module Bug

D. J. Morriss
Toronto, Ontario

. . .Under the wrong conditions, the bug configures the C-128 memory in unexpected ways. . .

A subtle bug exists in the C-128 Kernal routines for accessing the RAM Expansion Module, as found in the Version 0 ROM's (Listing 1). Under the wrong conditions, the bug configures the C-128 memory in unexpected ways, before transferring data to or from the RAM Expansion. Part of the bug has been corrected in the new Version 1 ROMs, and a software patch for Version 0 ROMs is supplied in this article. However part of the bug is still present, and indeed, seems to be an essential part of the operation of the RAM Expansion.

As a result of this bug, any particular RAM Expansion operation runs a small chance of accessing the C-128 Bank 15 configuration, no matter what configuration has been chosen. It doesn't matter whether the configuration has been set by the BANK command from BASIC, or by M/L operations on the Memory Management Unit. To the extent that Bank 15 differs from the chosen configuration, the RAM Expansion operation will cause one of several problems. If the bug acts during a STASH, the wrong data will be placed in the RAM Expansion. If the bug strikes during a FETCH, the data will be sent to Bank 15, and lost. In addition, it could overwrite important locations in Bank 15, crashing the system.

I encountered the bug while writing a BASIC program to sort a double sided disk full of strings; many more than could be loaded into the C-128 at one time. The brute force technique I used was to read as many strings as possible into the C-128 memory, sort them using a machine language routine, and then temporarily store the sorted strings in a bank of the RAM expansion. I stored the strings individually, using a routine like this:

```
10 DEF FNDK(X) = PEEK(X) + 256*PEEK(X+1)
   . . . . . .
   . . . . . .

100 BANK 1: SLOW
110 FOR K=1 TO NS
120 J = POINTER(A$(K))
130 STASH NL, FNDK(J+1),NL*(K-1),BK
140 NEXT K
```

where NS is the number of sorted strings in memory, NL is the length of a string (all the same), and BK is the number of one of the 8 RAM expansion banks.

The program continued loading and sorting portions of the total list, and STASHing them in different banks of the RAM expansion until the last string was read from disk. Then the sorted subsets were merged into a single, sorted, disk file by first FETCHing the first string from each RAM expansion bank used. The smallest of these was written to disk, and replaced by FETCHing the next one from that RAM expansion bank. The smallest was again written and replaced, and so on. This continued until the last string was read from each RAM expansion bank and written to disk.

This process required some preliminary work. To allow the RAM Expansion to access the C-128 Bank 1, it was necessary to set Bit 6 of $D506. This is the Ram Configuration Register of the Memory Management Unit (MMU). This also makes the 40 column screen useless. The second problem is that the Kernal DMA CALL routine needlessly insists on configuring memory to make the I/O block from $D000 to $DFFF visible before accessing the RAM Expansion (this has been changed in the Version 1 ROMs). Since any strings stored in Bank 1 in this address range would be invisible to the RAM Expansion, I lowered the top of the string pool (in $39 and $3A) to below $D000.

When I tested the program on a file containing fifteen thousand strings, ten characters long, I discovered that a tiny fraction of the sorted strings were corrupted when they came back from the RAM expansion. Naturally, this ruined the merging process. In fact, the strings were changed into chunks, ten bytes long, from the C-128 ROM's! I determined that the strings were corrupted on STASHing, and that the pieces of ROM that replaced the strings came from the same addresses as the strings that were corrupted. The STASH command was executing perfectly, except that it was STASHing from Bank 15, instead of Bank 1!

It was a stroke of pure luck that I was able to discover this. As you would expect, trying to interpret pieces of ROM as strings leads to a mess of graphic, cursor control and screen color characters. But one of the strings that showed up corrupted turned out to be part of the list of BASIC commands, and another came back as "STRING TOO"; part of one of the BASIC error messages! I knew that these strings came from the ROMs, and it was easy to use the MONITOR HUNT command to show that **all** the corrupted strings were sections of ROM.

There was no pattern to the corruption. Repeated runs on the same data produced different corrupted strings each time. Only a few, maybe two to five, were corrupted each time, but that was more than enough. During a short vacation trip, I left my C-128 working away, STASHing and FETCHing for 80 straight hours. The result was 630 STASH failures in 3,512,380 STASHes. This works out to a failure rate of 1 in 5575!

When I examined the various parts of the C-128 ROM's involved in the RAM Expansion commands, I found the cause of the problem. To understand the bug, you need to know a little about the operation of the RAM Expansion itself.

The RAM Expansion module contains its own computer, the Ram Expansion Controller (REC). As computers go, the REC is not too bright. All it does is move bytes. It has an instruction set of four commands! On the other hand, it is very good at what it does, moving bytes much faster than the C-128 CPU can. So, to operate the RAM Expansion, the C-128 first loads the REC registers with information about the location and number of bytes to be transferred, tells the REC Command Register, at $DF01, what to do, and then gets out of the way. The REC takes over, moves the data, and then returns control to the C-128. For a complete listing of all the REC registers, see the Transactor article "Commodore External RAM Expansion Cartridges", by Dale A. Costello, Vol #8, Issue #2, page 38.

However, the complex memory configurations of the C-128 presented a problem. To tell the REC to take over, you must load a value into the REC Command Register at $DF01. Clearly, to do this the I/O block must be visible. But suppose the memory you want to move is in some other bank. If you can see the REC, the REC can't see the memory to be shifted. If the REC can see the memory to be shifted, you can't tell the REC to operate. The chip designers got around this by coming up with a delayed action feature. In effect, you can store a value in the REC Command Register which says, "Here's what I want you to do, and I want you to do it **after** the next write to address $FF00". This address is a shadow register for the MMU Configuration Register (the Configuration Register is **not** the same as the RAM Configuration Register!). Writing a value there will configure memory in any fashion you want. With this feature you can instruct the REC, and then rearrange memory, before the REC takes over.

One last point. If you are going to start changing memory configurations in the middle of a M/L program, you better be sure that the program will still be there when the new configuration arrives. So this part of the program must be in the common RAM, from $0000 to $0400, which appears in all memory configurations. With this information firmly in mind, let's take a look at the disassembly of the various routines that manage the RAM Expansion, in Listing 1.

The BASIC routine, starting at $AA1F, evaluates the parameters of the BASIC command, and puts them in the appropriate REC registers from $DF02 to $DF08. It also places a number, $84, $85, or $86 in the Y register. This number is destined to be stored in the REC Command Register. Apart from telling the REC to STASH, FETCH or SWAP in Bits 0 and 1, these numbers all have Bit #7 set and Bit #4 cleared, setting up the delayed action feature. A curious point is that the three numbers have Bit 2 explicitly set. Bit 2 of this register is listed as reserved for future use. Finally, the number of the bank you want to access is stored in the X register, and the routine jumps to $FF50. This would also be the appropriate entry point for any M/L use of the RAM Expansion.

This location is the entry point for the DMA–CALL routine in the New C-128 Kernal Jump Table. As such, it simply jumps directly to the actual routine, at $F7A5 in Version 0 ROMs.

At $F7A5, the routine gets, from a table, the MMU Configuration value corresponding to the bank to be accessed by the REC. This value is then modified (Bit #0 is cleared) to create a **new** memory configuration in which the I/O block is visible. There is no need to do this. Possibly this code was written before the delayed action feature of the REC was created. This new MMU Configuration Register value is then unnecessarily copied to the X register, and the routine jumps to the RAM portion in the common RAM at $03F0. And now the bug is about to be activated.

The RAM routine first stores the current MMU Configuration Register value in the X register. The REC Command Register value, patiently waiting in the Y register, is finally stored in the REC. THE REC IS NOW PRIMED AND READY TO OPERATE AS SOON AS A WRITE OCCURS TO $FF00. Naturally, the next command writes the altered MMU Configuration Register value to $FF00, reconfiguring memory according to the last BANK command (sort of) and triggering the REC operation. After the REC is finished, the old MMU Configuration register value is restored, and the routine exits to BASIC. So where's the bug?

Consider what happens if (make that "when") an Interrupt ReQuest occurs **during** the four machine cycles of the STY $DF01 at $03F3. The STY $DF01 instruction is completed, and the REC is still primed and ready to go. But the IRQ routine takes over, and now the REC is triggered by a STA $FF00 instruction, at $FF22 in the IRQ handling routine. The value stored is zero, and memory is configured for Bank 15 during the REC operation. The delayed action feature is cleared by the REC after its operation, so that when the STX $FF00 at $03F6 is executed after the IRQ, no REC operation results.

The REC operation that occurs uses all the correct addresses, RAM Expansion bank, and number of bytes; these were stored in their registers earlier. The "only" error is that the transfer takes place to/from Bank 15, rather than the desired Bank.

The IRQ occurs 60 times a second; with a 1 MHz clock there is approximately 1 chance in 4167 that an IRQ will occur during the four cycles of the STY $DF01 command. The observed rate of 1 in every 5575 is almost exactly 75% of this theoretical rate. Perhaps the danger zone includes only three of the four machine cycles. Placing responsibility for the bug on the timing of an IRQ also explains the apparent random nature of the bug. The ugly fact remains that about one in every 5000 Ram Expansion operations, on a random basis, is compromised.

There are several solutions. By far the best is to buy the ROM Upgrade Kit for the C-128. The ROM routines for the Ram Expansion access have been changed (see Listing 2) to eliminate many of the problems. The I/O block is no longer activated every time, and the IRQ is masked to avoid the bug entirely. In addition, there is no need to modify the RAM Configuration Register yourself, if you are accessing Bank 1. The code from $CF80 to $CF8E saves the old value of this register, checks the Configuration Register table to see if your last BANK command used RAM in Bank 1, and modifies the RAM Configuration Register to suit; the original value is restored after the REC has acted. This means that the 40 column screen can be used while accessing Bank 1 with the RAM Expansion. The only sign of what is going on is a flicker, as the RAM Configuration Register is momentarily changed. There are other fixes as well, to many parts of the ROMs.

If the new ROMs don't appeal to you, there is a partial software fix. Fortunately, the problem is in a RAM routine only 13 bytes long. The cure involves adding a PHP and a SEI command at the beginning and a PLP command at the end, and there are three unused bytes at the end of the faulty routine! I added these three commands to the RAM portion of the DMA CALL routine, and eliminated the string corruption completely (see Listing 3). This short program could be entered using the built in Monitor assembler, and then BSAVEd:

```
BSAVE B0,"your name", P1008  TO P1023
```

BLOADing this binary file before any RAM Expansion operations would fix the bug. The I/O block would still be visible all the time; that part of the problem is in ROM.

Part of the problem has **not** been solved. Exactly the same bug will be triggered by a Non Maskable Interrupt. Since a NMI, by definition cannot be masked, there is no way to prevent the bug from striking. It is inherent in the delayed action feature of the REC. While NMIs are rare in ordinary programming, they are the heart and soul of RS-232 communications on the C-128. Reliable use of the RAM Expansion for other than Bank 15, during RS-232 communications, would appear to be difficult, if not impossible. This might not be as big a problem as it looks. Bank 15 uses RAM from "Bank 0" from $0000 to $4000. This includes the RS-232 buffers, the VIC text and bit-mapped screens and the large Applications Area from $1300 to $1C00. All of these can be moved bug free to and from the RAM Expansion during RS-232 operations. Any other areas should be moved into one of these areas, before being sent to the RAM Expansion.

One point is clear from the disassembly of the BASIC routine. The order of the arguments is the same for the STASH, FETCH, and SWAP commands, and that order is number of bytes to move, start address in C-128 memory, start address in RAM Expansion memory, and bank number in RAM Expansion. Several references give the order of these arguments incorrectly, listing the RAM bank number as the third argument and the RAM Expansion address as the fourth. A possible explanation lies in the fact that two Commodore references, while giving the correct syntax of the commands, list the arguments in a different order when defining them.

As far as I know, this is the first published discussion of this potentially serious bug. Since Commodore knew about the bug, and prepared new ROMs that correct it, their silence on the matter is troubling to those of us with Version 0 ROMs and a RAM Expansion.

Listing 1

COMPLETE STASH/FETCH/SWAP ROUTINE

VERSION 0 ROMS

```
aa1f  lda #$84    ;STASH entry  - NOTE -
aa21  jmp $aa2b   ;             - all numbers with
aa24  lda #$85    ;FETCH entry  - bit 7 set and
aa26  jmp $aa2b   ;             - bit 4 clear
aa29  lda #$86    ;SWAP entry

                  ;put future REC command register
aa2b  pha         ;value on stack
aa2c  jsr $8812   ;evaluate three parameters
aa2f  jsr $a845   ;of BASIC command
aa32  sty $df07   ;and store in
aa35  sta $df08   ;the proper REC
aa38  jsr $880f   ;registers.
aa3b  jsr $a845
aa3e  sty $df02
aa41  sta $df03
aa44  jsr $880f
aa47  jsr $a845
aa4a  sty $df04
aa4d  sta $df05
aa50  jsr $8809

aa53  cpx #$10    ;branch if error
aa55  bcs $aa65   ;in parameter evaluation.
aa57  jsr $a845   ;evaluate last parameter
aa5a  stx $df06   ;and store in REC
aa5d  pla         ;recover future REC command register
aa5e  tay         ;value and store in Y register
aa5f  ldx $03d5   ;get BANK command value into X reg
aa62  jmp $ff50   ;go to Kernal jump table
aa65  jmp $7d28   ;parameter error routine

ff50  jmp $f7a5   ;jump table entry

f7a5  lda $f7f0,x ;get MMU configuration value from
                   table based on BANK value in X reg
f7a8  and #$fe    ;modify it to enable I/O
f7aa  tax         ;copy to X register (WHY??)
f7ab  jmp $03f0   ;jump to RAM portion of routine
-------------------------------------------------

03f0  ldx $ff00   ;OVERWRITE X register with current
                   MMU configuration value
03f3  sty $df01   ;store Y register to REC command
                   register-- REC primed
03f6  sta $ff00   ;store new MMU configuration value
                   AND trigger REC operation
```

```
-------------------------------------------------
    STASH/FETCH/SWAP takes place
-------------------------------------------------

03f9  stx  $ff00    ;restore MMU configuration register
                     value
03fc  rts           ;all done so back to BASIC
-------------------------------------------------
```

Listing 2

COMPLETE STASH/FETCH/SWAP ROUTINE

VERSION 1 ROMS

(BASIC routine from $AA1F to $AA65 unchanged)

```
-------------------------------------------------
ff50  jmp $cf80     ;note new jump table destination
-------------------------------------------------
cf80  lda  $d506    ;get RAM configuration value
cf83  pha           ;and stack it
cf84  eor  $f7f0,x  ;set Bits 7 and 6 of RAM
cf87  and #$3f      ;Configuration Register to
cf89  eor  $f7f0,x  ;reflect Bank command value
cf8c  sta  $d506
cf8f  lda  $f7f0,x  ;get Configuration Register value from
                     table
cf92  tax           ;again WHY?
cf93  php           ;save status, including interrupt
cf94  sei           ;SET INTERRUPT MASK
cf95  jsr  $03f0    ;note JSR, not JMP
cf98  plp           ;restore previous interrupt status
cf99  pla           ;pull old RAM Configuration register
cf9a  sta  $d506    ;value and store
cf9d  rts           ;done, so back to BASIC
-------------------------------------------------
03f0  ldx  $ff00    ;same as in
03f3  sty  $df01    ;Version 0
03f6  sta  $ff00
03f9  stx  $ff00
03fc  rts           ;NOT to BASIC!
-------------------------------------------------
```

Listing 3

NEW RAM DMA CALL ROUTINE

FOR VERSION 0 ROMS

```
03f0  php           ;new command
03f1  sei           ;new command
03f2  ldx  $ff00    ;all moved two bytes
03f5  sty  $df01    ;higher in memory
03f8  sta  $ff00
03fb  stx  $ff00
03fe  plp           ;new command
03ff  rts
```

Commodore 128 Machine Language

Steve Punter
Mississauga, ON

. . .The 8510 processor, which is really a 6510 with the newest iteration of Commodore's memory management added, can only address 64K at any one given moment. . .

If you do most of your C128 programming using a high level language, like BASIC, you generally don't need to concern yourself with such trivialities as memory management. On the other hand, anyone who has delved into the machine language side of the C128 knows only too well that the memory layering used in this machine is not the easiest to work with. Drawing on my experience programming WordPro 128, I would like to impart to you some of the programming techniques I used, in hopes that they might come in handy.

I start out assuming that you have a reasonable understanding of the C128 and its inner workings, for I can't afford to go into the intricate detail needed to explain all of this to a rank beginner.

Before we begin, let's look at *how* Commodore has arranged the 128K of RAM and the 48, or so, kilobytes of ROM.

The 8510 processor, which is really a 6510 with the newest iteration of Commodore's memory management added, can only address 64K at any one given moment. To allow it access to the huge amount of RAM and ROM, selected portions of this 64K address space are set up in such a way that they can be told to access one of a variety of items. Think of each of the 4 sections of this 64K memory space as being 4 different elevators in a small building. Each can be independently moved to any floor of the building, allowing a variety of different combinations of access.

I'm not going to deal with the INTERNAL and EXTERNAL ROM sockets here, so the only three things concerning us which can appear in these memory segments are the first 64K of RAM, the second 64K of RAM, or the 48K of system ROM. In addition to that, there is also a small section of I/O space, which can be selectively brought in, or moved out.

The two 64K RAM spaces are called *banks*; bank 0 being the first 64K, and bank 1 being the second 64K. In addition to this, varying amounts of memory, ranging from none, on up to 32K, can be declared as "common". This refers to the fact that regardless of which of the two RAM banks you are in, access to the common area will always give you bank 0 RAM. This sort of addressing is advantageous in that coding which must be available AT ALL TIMES can reside there.

When first considering the arrangement of WordPro 128, I was stuck with the prospect of having the text in one bank, and a multitude of buffers in the other. Linking all that together, I had to have code which could run in EITHER bank, and here is where the deficiencies of the C128 MMU start to show.

Although routines are supplied in ROM to fetch or store data into either bank, they become unacceptably slow when called repeatedly. The 2Mhz mode of the C128 can help mask some of this inherent slowness, but in such applications as word processing, the speed problem is rather acute.

After much thought on the matter, a number of programming criteria became clear: Number one, ALL 64K of the addressable memory space should be, by default, RAM. Number two, the code must exist in both banks, with each bank of code accessing and manipulating the data stored there. And number three, the code must be able to call routines in the alternate bank without any difficult coding needed at the place of the subroutine call.

Each of these three criteria brought about unique, though curiously related, solutions. The first problem concerns the switching in of ALL RAM. Immediately, two difficulties arise; interrupts and KERNAL calls. The interrupts have been thoughtfully taken care of for us, and require no special care, but KERNAL calls are a different matter. No program can operate without making calls to the KERNAL, for even printing to the screen, or fetching from the keyboard, requires it. To make KERNAL calls, we would need to switch in the appropriate ROM before the call, and switch it out afterwards. This is not acceptable, as it gets very messy and complicated. If we just leave the KERNAL switched in all the time, we loose immediate availability of 16K of RAM, and must resort to switching that in and out every time we need it.

Clearly a better solution was needed; one which would give us immediate access to either the KERNAL or the RAM without having to actively switch it in or out. Although this sounds like a case of "wanting your cake and eating it too", there is a simple answer. All that is required is about 200 bytes at the top of each 64K RAM bank, and two small subroutines in the common memory area.

The Invisible KERNAL

The first step to creating an invisible KERNAL is to write a fairly short set of subroutines into your program. The two subroutines are as follows:

Subroutine # 1

```
kerncall  sta  safeloc
          pla
          sta  safeloc+1
          pla
          lda  #>kernback-1
          pha
          lda  #<kernback-1
          pha
          lda  #$ff
          pha
          lda  safeloc+1
          sec
          sbc  #3
          pha
          lda  $ff00
          sta  bankhold
          lda  #$0e
          sta  $ff00
          lda  safeloc
          rts
```

Subroutine # 2

```
kernback  pha
          lda  bankhold
          sta  $ff00
          pla
          rts
```

At the very beginning of your program, a routine should be included which will store, into each 3 byte location from $FF81 through $FFF3, a call to subroutine "KERNCALL". The following routine will do just that:

```
setkern   jsr  setkern1
setkern1  ldy  #$81
kern1     lda  #$20
          sta  $ff00,y
          iny
          lda  #<kerncall
          sta  $ff00,y
          iny
          lda  #>kerncall
          sta  $ff00,y
          iny
          cpy  #$f6
          bne  kern1
          lda  $ff00
          eor  #%01000000
          sta  $ff00
          rts
```

This rather odd structure, calling a subroutine IMMEDIATELY after itself, is just a quick and dirty way of having the routine execute itself twice. Note the at the end of each call, the RAM bank is "flipped", causing not only both banks to be processed, but the bank to return to its original state. Note also that this routine MUST reside in the common area also, else it will "bank" itself out, and crash the machine.

Now that all this is done, you can safely make ANY KERNAL call in complete confidence that the KERNAL ROM, as well as the I/O space, will automatically be swapped in and out for you. But how does it all work?

Let's follow the path of seemly innocent "JSR FFD2" call from within our running program. First of all, the return address of our original call is pushed onto the stack, then execution is transferred to location $FFD2. At this location, a JSR KERNCALL is encountered, causing the return address of this subroutine call to pushed on the stack on top of the previous one. The interesting thing to note about this return address is that it's EXACTLY two higher than the desired KERNAL call, and is a key factor to how the KERNCALL subroutine works.

Upon entering the KERNCALL subroutine, the accumulator is put safely away in a location called "SAFELOC", then the first byte is pulled off the stack and saved away in a location called "SAFELOC+1". This first byte from the stack is actually the low order part of our second return address, the one which is exactly two higher than our desired KERNAL call ($FFD2 in this example). The next byte off the stack is merely thrown away. It represents the high order part of the return address, but will *always* be $FF, and so we need not waste any time, or memory space, storing it.

Next, we push the address of our second subroutine onto the stack in place of the original one that was generated when the JSR KERNCALL call was made. The purpose of this exercise is to cause the RTS within the KERNAL subroutine itself to transfer execution there, so that RAM may be restored. With me so far?

The next step involves the pushing of yet a third subroutine return address onto the stack, but the purpose of this is to use an RTS like a JMP INDIRECT. In other words, we now wish to set up a stack configuration that will cause an RTS to transfer execution to the desired KERNAL subroutine. First, we push the high order part of the address, which, as was noted earlier, is *always* an $FF. After that, we recall the low order part of this address and subtract three from it. Why three? After all, I did say that this address was exactly *two* higher than the desired KERNAL call. It's done simply because the address stored on the stack for an RTS is always *one byte* lower than the address at which execution actually begins.

Now we are ready to bank in both the KERNAL ROMs and the I/O space, but before we do that, a copy of the *current* memory configuration is saved in a location called "BANKHOLD". A value of $0E is then stored in the memory configuration register, and an RTS is executed. Since we have previously pushed the appropriate address onto the stack, the RTS causing execution to

go *directly* to the KERNAL call we originally made, or $FFD2 in this case. Just before the RTS is executed though, the value of the accumulator is restored so that the proper value is passed to the KERNAL routine.

Once the KERNAL subroutine has finished doing whatever it is that it's supposed to do, its RTS causes execution to be transferred to our "KERNBACK" routine. The reason for this is that we deliberately pushed this return address onto the stack for this very reason. KERNBACK's purpose is very simple; it merely restores the memory configuration register to its former value (as saved in the just before jumping to our desired KERNAL call). The accumulator is appropriately saved so that KERNAL routines which pass values out are not disturbed.

As the RTS of KERNBACK is reached, execution is passed back to the code immediately after the actual KERNAL call (our $FFD2 example). Although the process may seem complicated, it's not hard to see how it *does* let you have your cake and eat it to.

Multiple BANK Coding

The second major problem was to design a system whereby a single program could exist in two separate banks of RAM. This presents a problem that is very similar to the KERNAL routines, and is solved in much the same way.

Before you can begin to write such a program though, certain basics must be considered, and a "game plan" must be formulated. There are three distinct areas of memory in which your program code will reside: common RAM; bank 0 RAM; and bank 1 RAM. What goes where, and how much common RAM to assign, is what we have to decide.

In WordPro 128, it was decided that the text would reside in bank 1, while all other necessary buffers and strings would be placed in bank 0. Since it was desirable to maximize the amount of text, code in bank 1 had to be kept to a minimum. Bank 1 would obviously contain code which *only* made direct references to text.

Bank 0 could contain just about everything else, with the exception of two special cases: routines which were called *quite often* from code in *both banks*; and routines which made reference to both text AND buffers. This code will have to go in the common area.

Since WordPro 128 is a huge program, it was decided that the bottom 16K of RAM would be deemed common, leaving 48K in each of the banks for other purposes. Your applications may not call for such a large expenditure of common RAM. How much you need is not something I can tell you in this article, it's really a matter of looking at your needs and deciding for yourself.

Since common code will reside from $0000 to $3FFF, bankable RAM begins at $4000. All that is left now is to work out a way of allowing routines in one bank to call routines in the other. As I hinted at earlier, the solution is very similar to the one used to leave the KERNAL ROM banked *out* until needed. The solution to this second problem starts out with a jump table to all the necessary routines at the beginning of *each* of the two 48K banks of *non*-common RAM. The trick to all of this lies in *what* we put at each of the jump table locations.

If the routine represented by a particular jump table location resides in the current bank, then a simple JMP to it is used. Should it reside in the *other* bank, a JSR call is placed there to a routine very much like the one used to bring in the KERNAL ROM. If this sounds a little complex, let's take a small example. There are four subroutines, called "SUB1", "SUB2", "SUB3" and "SUB4". Subroutines 1 and 2 are in bank 0, while subroutines 3 and 4 are in bank1. This is how the four jump table locations would appear at the beginning of each bank:

* In BANK 0:

```
        jmp  sub1
        jmp  sub2
sub3    jsr  swapbank
sub4    jsr  swapbank
```

* In BANK 1:

```
sub1    jsr  swapbank
sub2    jsr  swapbank
        jmp  sub3
        jmp  sub4
```

Were a routine in bank 0 to call "JSR SUB1" or "JSR SUB2", they would branch directly there as if no banking existed. A routine in bank 1 calling "JSR SUB3" or "JSR SUB4" would have the same effect. But, a routine in bank 0 calling "JSR SUB3" or "JSR SUB4", or a routine in bank 1 calling "JSR SUB1" or "JSR SUB2" would be directed to the "JSR SWAPBANK" call in the jump table. SWAPBANK is very much like KERNCALL, but it swaps around which of the two non-common banks of RAM are currently active, then jumps back to the *same* location in the jump table (just like the KERNCALL routine). Now that the bank has been switched, it runs into the appropriate JMP instruction, rather than the JSR SWAPBANK.

Of course, while in this alternate bank subroutine, a call *back* to the original bank might be made, so a "stack" of return "banks" must be implemented so that each subroutine is returned to it's correct bank.

Here are the two subroutines involved in this task:

Subroutine # 1:

```
swapbank  sta  safeloc

          tya
          pha
          ldy  bankpnt
          lda  $ff00
```

```
                and  #$fe
                sta  bankstak,y
                inc  bankpnt
                pla
                tay
                pla
                sbc  #3
                sta  safeloc+1
                pla
                sbc  #0
                sta  safeloc+2
                lda  #>bankback-1
                pha
                lda  #<bankback-1
                pha
                lda  safeloc+2
                pha
                lda  safeloc+1
                pha
                lda  $ff00
                eor  #%01000000
                sta  $ff00
                lda  safeloc
                rts
```

Subroutine # 2:

```
bankback  php

                pha
                tya
                pha
                dec  bankpnt
                ldy  bankpnt
                lda  $ff00
                and  #$01
                ora  bankstak,y
                sta  $ff00
                pla
                tay
                pla
                plp
                rts
```

When a routine makes an alternate bank call, the return address of the subroutine is pushed onto the stack, then execution is transferred to the jump table, where a JSR SWAPBANK call is made, thus pushing the return address of this routine onto the stack also. As with the KERNCALL routine, this address is exactly two higher than that of the jump table address just referenced. Execution now branches to the SWAPBANK routine in common RAM, at which point, the accumulator is saved in "SAFELOC". The Y-register is pushed onto the stack so that its value is preserved while we push the memory control register onto our special "bank stack".

Once the bank has been pushed on our stack, the Y-register is restored, and the next two bytes are pulled from the stack. As previously noted, they contain a pointer to the appropriate jump table location, but two higher. Subsequently, three is subtracted from this 16 bit value and the whole thing is stored in "SAFELOC+1" and "SAFELOC+2". The reasons for this were discussed previously in the section on "The Invisible KERNAL". In place of this return address, we push the location of "BANKBACK", which, like "KERNBACK", is responsible for putting everything back to normal after the desired routine has finished.

Next, we push the value of the jump table location (minus one) onto the stack so the next RTS acts like a JMP INDIRECT and transfers execution *back* to the same location in the jump table that we came from, only this time, the banks have been switched, so there will be JMP instruction there, not a JSR SWAPBANK. Before reaching the RTS though, the appropriate bit in the memory control register is "flipped" so that the *other* 48K of non-common RAM is brought in. The accumulator is restored, and the RTS performed.

Once the desired subroutine has finished, its RTS transfers execution to our "BANKBACK" routine, which basically pulls the correct memory configuration byte from our special stack so that when we return to the calling routine, the correct bank configuration will be in force.

Observant readers will notice that bit 0 of the memory configuration register is *not* being saved on the special stack. This bit controls the existence of the I/O space, and for most purposes, I decided that it would desirable to leave it unchanged so that I/O control stays within the *mainline* coding.

Even more observant readers will notice that the special stack isn't really needed, since for a routine to call this function, it *must* have been in the alternate bank. Logic would dictate, therefore, that to return control to the calling routine would require nothing more than swapping active banks again. Although this is true, the bank stack guarantees that no screw ups within the called routine cause the bank settings to become misaligned.

Some Last Words

Of course, the two solutions I've presented here don't even begin to scratch the surface of what M/L in the C128 is all about, but they should give you a good starting place so that programming is that much easier.

Many may argue that M/L is a dead practice, since so many high level languages, especially C, allow you to do almost the same thing with much greater ease. Although I fully agree that languages like C are very powerful, they simply can't match the grace and speed possible by writing your programs in pure machine language.

Given a choice, I usually opt for machine language, but also recognize that certain tasks are far more suited to a higher level language. Before you tackle a task in M/L on the C128, you should decide this too.

Auto Booting CP/M Programs On The C-128

Miklos Garamszeghy
Toronto, ON

. . .There are two methods. . . the first is fairly well known. . .
The second is not well known at all, even to confirmed hackers. . .

There are two ways to auto boot applications programs, utilities, games, etc. on the C-128 in CP/M mode. The first is a fairly well documented feature using a special SUBMIT file. Each time CP/M is started up, it looks for a file named "PROFILE.SUB" on the default drive. This is a regular CP/M submit file which is automatically executed on start up, similar to the IBM-PC-DOS "AUTOEXEC.BAT" file. Unlike the IBM version, however, PROFILE.SUB requires that you have the utility "SUBMIT.COM" on the same disk. This utility reads the statements in the SUBMIT file (standard CP/M commands or program names such as DIR or PIP etc.) and translates them into its own executable form. The commands are not executed immediately, but are written to a temporary file called "SYSIN.$$$". When all commands have been translated and written to this file, the file is read and they are executed.

This brings out two limitations of the PROFILE.SUB method of auto booting. Because a temporary file is written to the disk, you cannot cover the write protect notch. The method is also quite slow, especially with a 1541 drive, because it involves reading and writing a number of disk files. In addition, since the PROFILE.SUB file is written in standard ASCII format, it is very susceptible to prying eyes. All it takes is a "TYPE PROFILE.SUB" command for some one to sneak a peak at your secret boot routine. Despite these limitations, you can do a very complicated set of start up procedures using this method.

The second method is not well known at all, even to confirmed hackers who have been using other CP/M machines for years. (One of the beauties of CP/M is that it is very transportable between machines. If it works on one machine, it will most likely work on other CP/M machines.) Although not quite as simple as using the PROFILE.SUB file, the method allows you to cover the write protect notch. In addition, you do not need any other files on the disk other than the one you want to boot. It is also much faster, because it does not have to read and execute the SUBMIT.COM program before executing the boot. The nature of the technique limits you to a single CP/M command in the boot statement, but this is sufficient to start up most programs. The technique involves changing a few bytes in the CPM Console Command Processor (CCP.COM) file. (Because you are changing the system files, never do this on your original disks! Always work with a backup copy.)

The CCP is the part of the CP/M system which reads and interprets commands typed in at the console. It contains the code for accessing most of the built-in CP/M commands, such as DIR, TYPE, etc. It also loads and transfers execution to transient program files. Control of the system is passed to CCP.COM after boot up and after each command or transient program has finished executing. Using a debugging utility, such as SID.COM supplied on the CP/M+ Additional Utilities Disk, (or the older CP/M debugger called DDT.COM or one of the many public domain utilities), load in the file with the command:

```
SID CCP.COM <return>
```

SID will respond with:

```
NEXT  MSIZE  PC    END
0D80  0D80   0100  D2FF
#
```

The # is the SID prompt. The bytes of interest start at hex address $07C2. Use SID's display command to display the memory starting at that location:

```
D07C2 <return>
```

(note that there is no space between D and the address) The first line of the display should look like:

```
07C2:53 55 42 4D 49 54 20 20
          43 4F 4D 1A 06 4B CD F9:SUBMIT  COM. . . . .
```

By replacing the word SUBMIT (i.e. the bytes from 07C2 to 07C7) with ASCII spaces (hex $20s), you can autoboot a program file on start up. This is done with SID's "S" command:

```
S07C2 <return>
```

SID responds with:

```
07C2 53 ⌖ (where ⌖ represents a flashing cursor)
```

Type in a quote mark at the cursor followed by 5 spaces and a <return>:

"<5 spaces> <return>

SID responds with:

07C8 20 ⌖

Type in a period followed by <return> at the cursor to end the substitution:

. <return>

SID will respond with the prompt #.

This change makes CP/M look for the file PROFILE.COM when it starts up. If this file is present, it will automatically load and execute it. All you must do to take advantage of this is to rename your program to "PROFILE.COM" with CP/M's "RENAME" command. For example, to boot MBASIC you would use:

RENAME PROFILE.COM = MBASIC.COM <return>

Of course, there is nothing sacred about the name "PROFILE". If you would rather not rename your program, you can make an additional change to the CCP. Addresses $04FC to $0503 contain the word "PROFILE". You can change this to your application program name using a similar method to that outlined above for erasing the SUBMIT command. For the MBASIC example, you would use:

you type	S04FC <return>
SID responds	04FC 50 ⌖
you type	"MBASIC <space> <return>
SID responds	0503 2E ⌖
you type	. <return>
SID responds	#

The total length of the string must be seven characters so pad the end of the filename with spaces.

Once you have made all of the necessary changes, save the modified file back to disk with the command:

WCCP.COM,0100,0D80 <return>

The 0100 and 0D80 are the hexadecimal start and end addresses of the CCP.COM file as listed on the SID startup display. All hex addresses listed above are identical for all current versions of CP/M+ implemented on the C-128.

Now press the reset button and watch your program boot up automatically! One final note. If you decide to change the boot file name from "PROFILE.COM" to something else, you will have to modify a custom CCP for each disk you want to boot. Of course, only one boot routine per disk is possible with this routine. If you format a new disk and copy the modified system files to it, the boot command will be totally transparent if the bootable file is not present on the new disk. This means that if the file mentioned in the boot routine is not present on the disk, CP/M will start up in its normal manner and end with the familiar A> prompt.

Clock–Calendar 128

William J. Brier
Bensenville, IL

An increasingly popular accessory for personal computers is the Clock–Calendar cartridge with battery backup. It is quite handy to have the time and date readily available when you or the computer wants it. However, such gadgetry requires the use of that elusive matter form known as money. Not surprisingly, most of us hate spending money (unless it's somebody else's).

I needed some way to display the time and date on my C-128 but I didn't need the battery backup (or the expense). If you have read this far, you probably share the same needs. So, if you are willing to live without the battery backup feature and are also willing to do some typing, I have an inexpensive timekeeping solution for you. It uses some resources that all C-128 computers are equipped with, namely the TOD clocks in the CIA chips. My solution is a little machine language utility called (oddly enough) Clock–Calendar 128.

Clock–Calendar 128 is a time and date utility for the C-128. Clock–Calendar uses the Time Of Day (TOD) clock in CIA #2 to generate an accurate 24 hour time output, with a resolution of one second. The utility also maintains a Gregorian calendar, accurate through the year 2099. Time and date outputs are made available for display on both 40 and 80 column screens and for use by other software.

In addition to the time and date functions, Clock–Calendar also features an audible alarm, which may be set to any desired time. All functions of Clock–Calendar are driven by interrupt requests and are therefore transparent to the computer's operating system and most software. Special software traps prevent interference with the C-128 screen editor when the programming considerations described later in this article are observed.

It's Not A Hydrogen Maser Standard But. . .

Unlike the TI$ software "clock" provided as part of BASIC 7.0, the time output of Clock–Calendar is remarkably accurate, typically demonstrating a drift of less than 0.5 seconds per month. This is because the CIA hardware clock is synchronized to the power line frequency, which is carefully regulated by the utility that supplies electric power.

The CIA hardware clock is initialized during the reset sequence to operate on a 60 Hz power line frequency (the North American standard). Many European and Middle East locations use 50 Hz power. A simple POKE (described later) can be used to program the CIA chip to operate on 50 Hz power.

I must caution you that batteries are not included with Clock–Calendar. As soon as you (or the power company) shut off the C-128 the clock vanishes into thin air (sort of the way time does when you're behind schedule).

Clock–Calendar was inspired by a program that was published for the C-64 by Mike Forani in Volume 5, Issue 2 of TRANSACTOR. I typed in Mike's program (learning something about interrupts in the process) and later added the calendar and alarm routines to it. When I acquired my C-128 I decided to port the program over, accommodating both screen displays in the process. That porting process proved to be a bit of an education in itself, especially in getting the clock display and the C-128 screen editor to get along with each other. I ended up starting from scratch, although Mike may recognize a slight vestige of his original routine. In the process of doing this rewrite I found that a programmer's best friend is indeed the RESET button.

Three Modules And No Batteries

Clock–Calendar consists of three machine language modules (one main module and two setup modules). I have this annoying habit of giving binary files cryptic, "computer-sounding" names that tell me what they are. Accordingly, I've given each of the three modules cryptic, computer-sounding names that will undoubtedly annoy you for the balance of this article:

CLK4864
CLK5632
DT5632

The CLK4864 module contains all program instructions required to execute the functions of Clock–Calendar. This includes time and date decoding functions as well as the audible alarm feature. CLK5632 is used to set the time of day or the alarm time, while DT5632 is used to set the displayed date. The numbers in the filenames indicate the decimal addresses to which each file loads into memory.

Because these are all binary files you must load them into memory with the BLOAD syntax (or LOAD"FILENAME",8,1). Read on to discover how to get Clock–Calendar running. As you'll see, it's actually quite user-friendly (honest!).

Placing Clock–Calendar Into Operation

To activate Clock–Calendar simply BLOAD the CLK4864 module into memory and activate it with SYS 4897. This command will "wedge" Clock–Calendar into the interrupt system and the time and date display will appear in the upper right-hand corner of both display screens.

When Clock–Calendar is placed into operation the hardware clock is given a "nudge" to get it started (it initially is not running when the C-128 is powered up or after the RESET button has been pressed). Therefore, it is probable that the displayed time will not be

correct. Also, the displayed date will be represented by a series of question marks (??-??-??) indicating that no date has been set.

Upon activation of Clock-Calendar the default display settings will be in force. These are yellow display colour on the 40 column screen, cyan display colour on the 80 column screen and normal (non-reversed) video display on both screens. Provisions have been made to change the display characteristics to suit individual tastes. Such adjustments are described in a later section.

One precaution must be noted before leaving this section. Once you have loaded and activated Clock-Calendar do not repeat the loading process. Doing so will probably result in a system crash. This of course, will give you an opportunity to make friends with the RESET button, in case you haven't already done so.

Setting The Time Of Day

Before describing how to set the time, it is appropriate to discuss the manner in which Clock-Calendar displays the time.

The time will always be displayed in the format HH:MM:SS, where HH is the hours representation, MM is the minutes representation and SS the seconds representation. The time is kept in 24 hour or military format. This means that when the time of day is 1:00 PM through 11:59 PM, the clock will display the time with the value 12 added to the hours. Midnight (12:00 AM) will be displayed as 00:00:00. The main advantage of the 24 hour time-keeping system will quickly become apparent: There will never be any confusion regarding AM or PM.

To set the time of day, proceed as follows:

1. BLOAD the CLK5632 module into memory.
2. Type SYS 5632,0. The '0' indicates to CLK5632 that you are setting the time of day.
3. The C-128 will prompt you for the CORRECT TIME (HH:MM) and flash the cursor. Type in the time of day using 24 hour format. Note that you must type two digits for the hours and two digits for the minutes. Enter midnight as 00:00. The cursor will skip over the colon as you enter the time (see, I told you it was user-friendly).
4. Upon typing in the time, press RETURN to enter it or the DELete key to erase your input.

NOTE: The time value that you enter must be a valid one. The hours value may not be greater than 23 and the minutes may not be greater than 59. If you don't adhere to these requirements, Clock-Calendar will become unfriendly and demand that you do it right.

5. Next, the time that you have entered will be confirmed for accuracy. Respond to the confirmation prompt with (Y)es or (N)o. A (N)o terminates entry without affecting the time setting. In either case, control will be returned to the calling program or BASIC.

Setting The Alarm Time

As mentioned above, Clock-Calendar incorporates an audible alarm function. The alarm is activated when the time of day equals the alarm time and the alarm enable flag (described later) has been set to an enable value.

Set the alarm as follows:

1. BLOAD CLK5632 into memory.
2. Type SYS 5632,1. The '1' indicates to CLK5632 that you are setting the alarm time.
3. You will be prompted for the ALARM TIME (HH:MM). The alarm time is entered in exactly the same manner as the time of day.
4. As with entering the time of day, responding with (N)o to the confirmation prompt will abort the operation with no effect on Clock-Calendar. Otherwise, the alarm time will be stored and the alarm enable flag will be set.

Upon reaching the alarm time a "gong" will sound once every two seconds for a period of one minute. When the one minute period has elapsed the gong will silence. Unless the alarm enable flag is cleared, the alarm will sound at the same time each day for as long as Clock-Calendar remains in operation.

To silence the alarm and disable it, POKE 4896,129. To enable the alarm again so that it chimes the next day at the same time, POKE 4896,0.

Setting The Calendar Date

Before describing how to set the date, it is appropriate to discuss the manner in which Clock-Calendar displays the date.

The date is displayed in the format MM-DD-YY where MM is the month, DD is the day and YY is the year. If the month is January through September the first digit of the date will be blank. The date is always displayed immediately below the time in the upper right-hand corner of the screen.

The date automatically changes at the stroke of midnight provided that Clock-Calendar is in operation. Because the calendar is maintained in software the date will not change if the program has been disabled.

Clock-Calendar incorporates means to detect a leap year and make appropriate compensation for the month of February. This compensation is accurate for all years from 1901 to 2099 inclusive. Inaccuracies will occur in any century year that is not evenly divisible by 400, as Clock-Calendar does not use the first two digits of the year (for example, 1900, 2000 and 2100 are all the same to Clock-Calendar but only the year 2000 is a leap year).

To change the date, proceed as follows:

1. BLOAD"DT5632".
2. Type SYS 5632.
3. The C-128 will prompt you for TODAY'S DATE and flash the cursor. Type in the date as MM-DD-YY. If the month is January through September type a zero as the first month digit.
4. Similarly, days one through nine and years zero through nine must be entered with a zero as the first digit. The cursor will automatically skip over the hyphens (-) as you enter the date.

NOTE: The date value that you enter must be a valid one. The month must be 1 through 12, the day must be 1 through 31 and must also be a valid day for the month and year that has been entered. For example, entering 02-29-87 would not be valid as 1986 was not a leap year. Similarly, entering 11-31-87 would also

not be valid as there are only 30 days in November.

5. Next, the date that you have entered will be confirmed for accuracy. A (N)o response will simply abort without affecting the displayed date.

Modifying The Clock–Calendar Display

As described above, Clock–Calendar has default display settings that determine the colour and appearance of the time and date display. It is possible by use of simple POKEs to change the appearance of the display to reverse video, to change the display colours or to discontinue updating of the display. These functions are controlled by a group of memory locations starting at $1315 in RAM 0. Their functions are as follows (all location addresses are given in hexadecimal):

$1315 This location is the "display flag". Clock–Calendar will display the time and date on the screen as long as the display flag is set to zero. If you wish to freeze the display POKE a 1 into this location. This will tell Clock–Calendar to stop updating the display.

No other function of Clock–Calendar will be affected in any way. The display flag is set to 0 when the program is activated.

$1316 This location determines the display colour on both screens. If you wish to change the display colours POKE any value from 0 to 15 into this location. The proper value may be selected from the following chart:

Code	40 COL.	80 COL.	Code	40 COL.	80 COL.
0	Black	Black	8	Orange	Dark Red
1	White	Dark Grey	9	Brown	Light Red
2	Dark Red	Dark Blue	10	Light Red	Dark Purple
3	Cyan	Light Blue	11	Dark Grey	Light Purple
4	Purple	Dark Green	12	Medium Grey	Dark Yellow
5	Dark Green	Light Green	13	Light Green	Light Yellow
6	Dark Blue	Dark Cyan	14	Light Blue	Light Grey
7	Yellow	Light Cyan	15	Light Grey	White

Colour values in excess of 15 are undefined in the program and may cause the display to behave in unpredictable ways.

$1317 This location determines whether the display will be in normal or reverse video. POKEing a 1 into this location will change the display to reverse video. The default setting is 0 (normal video).

Using Clock–Calendar With Other Software

Clock–Calendar has been designed so that it may be utilized with other software that requires a time and/or date value. Because Clock–Calendar is interrupt-driven it is "invisible" to BASIC and most software and therefore requires few special programming considerations. There are a few precautions that must be observed when using the 80 column screen or when fetching keyboard input by use of the INPUT statement in BASIC or the CHRIN (BASIN) subroutine in the Kernal. These precautions will be discussed later in this section.

The time and date outputs of Clock–Calendar are available in a group of memory locations starting at $1300 in RAM 0. These locations are updated once per second as long as Clock–Calendar remains activated. The actual time and date values are stored in consecutive byte order, using both PETASCII and binary-coded decimal (BCD) format. A description of each output follows in location order:

$1300 PETASCII Date
The PETASCII form of the date is stored starting at $1300 and takes the form MM–DD–YY terminated by a CHR$(0) character. The date changes only at midnight (when hours, minutes and seconds all equal zero).

Because it is possible for the date to change while your software is reading it, you should "stop the clock" by issuing a SYS 4900 command immediately before fetching the date. Once the fetch has been completed restart the clock with SYS 4897.

NOTE: If the display flag at $1315 has been set to "display off" (1) it will be reset to "display on" when the SYS 4897 command is issued.

A recommended method of fetching the date from BASIC is as follows:

```
100 dt$ = "": i = dec("1300"): sys 4900: for j = 0 to 7
110 dt$ = dt$ + chr$(peek(i + j)): next: sys 4897
```

This will result in the current date being assigned to variable DT$.

If the month is between January and September the first character in the string will be CHR$(32) (a blank). If the string is written to a tape or disk file the leading blank will be stripped when INPUT# is used to read it back.

$1309 PETASCII Time
The PETASCII form of the time is stored starting at $1309 and takes the form HH:MM:SS terminated by a CHR$(0) character. The time changes once per second and ranges between 00:00:00 (midnight) and 23:59:59 (one second before midnight).

As with fetching the date you should stop the clock immediately before fetching the time. A recommended method of fetching the time is as follows:

```
120 td$ = "": i = dec("1309"): sys 4900: for j = 0 to 7
130 td$ = td$ + chr$(peek(i + j)): next: sys 4897
```

This will result in the current time being assigned to variable TD$.

NOTE: The time string will contain two imbedded colons (:). If you write the string to a tape or disk file you must read it back with the GET# statement.

$1312 BCD Date
The BCD form of the date is stored starting at $1312 and takes the form M D Y in three consecutive bytes. If your software requires storage of the date into a file you may find the BCD form more efficient as less storage space is required.

A recommended method of fetching the BCD date is as follows:

```
140 bd$ = "": i = dec("1312"): sys 4900: for j = 0 to 2
150 bd$ = bd$ + chr$(peek(i + j)): next: sys 4897
```

This will result in the BCD date being assigned to the variable BD$. You may PRINT# this date to a file and read it back with INPUT# as there will be no embedded PETASCII values that will cause trouble with INPUT#.

Using BASIC 7.0 it is possible to decode the BCD date into a PETASCII representation suitable for display. A recommended method of doing so is as follows (the BCD date is assumed to be in variable BD$):

```
200 m = asc(mid$(bd$,1))
210 d = asc(mid$(bd$,2))
220 y = asc(mid$(bd$,3))
230 m$ = mid$(hex$(m),3):if val(m$) < 10 then
    mid$(m$,1,1) = chr$(32)
240 d$ = mid$(hex$(d),3) 250 y$ = mid$(hex$(y),3)
260 dd$ = m$ + "-" + d$ + "-" + y$
```

This will result in the variable DD$ containing the PETASCII date, with a leading blank if the month is January through September.

$1318 Signature

This location contains the PETASCII string "CLK" terminated with a CHR$(0) character. The signature is present only when Clock–Calendar is activated. Applications software may use this string to determine if the clock is running. The recommended method of doing so is as follows:

```
270 cs$ = "clk" + chr$(0): ck$ = "": ss = dec("1318")
280 for i = 0 to 3: ck$ = ck$ + chr$(peek(ss + i)):next
290 if ck$<>cs$ then CLOCK IS NOT ACTIVATED
```

It is good practice to always test for the signature string before attempting to fetch time and/or date values from the clock.

$1327 BCD TIME

The BCD form of the time is stored starting at $1327 and takes the form H M S in three consecutive bytes. If your software requires storage of the time into a file you may find the BCD form more efficient as less storage space is required.

A recommended method of fetching the BCD time is as follows:

```
300 bt$ = "": i = dec("1327"): sys 4900: for j = 0 to 2
310 bt$ = bt$ + chr$(peek(i + j)): next: sys 4897
```

This will result in the BCD time being assigned to the variable BT$. You may PRINT# this time value to a file. However, it must be read back with GET# to avoid a truncated string.

Using BASIC 7.0 it is possible to decode the BCD time into a PETASCII representation suitable for display. A recommended method of doing so is as follows (the BCD time is assumed to be in variable BT$):

```
320 h = asc(mid$(bt$,1))
330 m = asc(mid$(bt$,2))
340 s = asc(mid$(bt$,3))
350 h$ = mid$(hex$(h),3)
360 m$ = mid$(hex$(m),3)
370 s$ = mid$(hex$(s),3)
380 td$ = h$ + ":" + m$ + ":" + s$
```

This will result in the variable TD$ containing the PETASCII time, with a leading zero if the hour is before 10:00 AM.

To avoid system clashes or other maladies when using Clock–Calendar with other software, you should observe the following precautions:

1. Do not disturb anything in RAM 0 between locations $131C and $15D8 inclusive (except the alarm locations at $131E–$1320). Because Clock–Calendar is driven by interrupts it may be considered part of the C-128 operating system. Disturbing anything in the $131C–$15D8 memory range will probably cause the C-128 to immediately crash. If it is necessary to load a different program into that same area you must stop the clock with a SYS 4900 command before attempting the load.

2. Avoid using the INPUT statement in BASIC (or CHRIN in machine language) to fetch keyboard input when the cursor is on the top two rows of the screen. The system will accept the time or date display as part of the input. If you must use the top two rows for input, either define a window that does not include the columns in which the time and date display occurs or else shut off the display as previously described.

NOTE: Shutting off the display does not remove it from the screen. It simply tells Clock–Calendar to discontinue updating of the display. You must explicitly clear that part of the screen.

3. When using INPUT or CHRIN on the 80 column screen it is mandatory that the display be shut off, regardless of the row on which the input is to be accepted. Due to the manner in which the C-128 screen editor operates, a clash between CHRIN and Clock–Calendar may occur if the INSerT/DELete key is used or if the screen is scrolled with the cursor keys. Such a clash may result in a scrambled screen display, from which recovery requires use of the STOP/RESTORE keypress combination.

   It is also not advisable to use the SCNCLR statement from BASIC when operating on the 80 column text screen. SCNCLR bypasses software traps that have been built into Clock–Calendar to avoid clashes with the screen editor. Instead, use the C-64 method of clearing the screen (PRINT CHR$(147)). Partial screen clearing by use of the ESCape functions (ESC P, ESC Q and ESC @) is permissible.

4. If your program makes use of split-screen graphics you should stop the clock with a SYS 4900 command. Split-screen graphics require very precise timing of the interrupts. The small additional time required to execute the Clock–Calendar code is sufficient to disturb that timing. Although such a disturbance will not be fatal to the system it may result in an unattractive display.

5. If you are using a music program you should avoid setting the alarm. The alarm function, of course, uses the SID chip and will interfere with any music that is being performed. Also, many music programs make use of synchronized graphics, such graphics being timed by interrupts. In such a case, the clock should be stopped for the reasons described above.

6. If you are programming in machine language, avoid direct jumps into the screen editor ROM unless interrupts have been disabled. Direct ROM jumps will bypass the editor traps built into Clock–Calendar and may cause the 80 column display to malfunction while interrupts are enabled. This precaution does not apply to the 40 column screen.

Controlling The Alarm

As mentioned above, Clock–Calendar features an audible alarm. The alarm is controlled by both the alarm time setting and the alarm enable flag at $1320. Setting the flag to any positive value (0–127) will enable the alarm while setting the flag to any negative value (128–255) will disable the alarm and immediately silence it if it is operating.

NOTE: Even though the alarm enable flag has been set to "alarm on" a valid alarm time must exist in the alarm time location if the alarm is to function.

If you do not disable the alarm with the flag it will automatically silence after one minute.

If it is necessary to determine what the current alarm time is you may decode the alarm setting from BASIC 7.0 using the following short routine:

```
400 i=dec("131e"): h$=mid$(hex$(peek(i)),3)
410 m$=mid$(hex$(peek(i+1)),3): at$=h$+":"+m$
```

This will result in the decoded alarm time being assigned to variable AT$ in the format HH:MM (seconds are not used for the alarm time). It is not necessary to stop the clock to fetch the alarm time as it is used only as a reference by Clock–Calendar.

Adjusting The Hardware Clock For 50 Hz Operation

As mentioned above, the CIA hardware clock is initialized to use 60 Hz as a time-keeping reference. If your power line operates at 50 Hz you may make the necessary adjustment with the following command sequence:

```
i=dec("dd0e"): bank15: poke i,peek(i) or 128
```

If it is necessary to reset it for 60 Hz power you may use the following command sequence:

```
i=dec("dd0e"): bank15: poke i,peek(i) and 127
```

If the CIA chip is set for the wrong power line frequency it will be obvious by watching the speed at which the display updates.

More Than Just a Clock

Clock–Calendar takes care of the basic functions of timekeeping and, as you'll see, does the job at minimal cost. However, that doesn't mean that the program can't be put to other uses.

I've purposely placed the BCD time and date outputs in stable locations so that they may be used by other software. I've also defined a stable exit location (IRQA) for the program so that a machine language programmer can use Clock–Calendar with a non-standard IRQ system. This makes it possible to easily patch into the program for other purposes.

In a future article we could explore such things as controlling external devices with Clock–Calendar or perhaps getting the C-128 to automatically execute a program at any desired time. Before you know it, we'll have your computer running the house and making coffee in the morning. Now, that's what I call user-friendly!

How Clock–Calendar Works

Since this is a magazine article and not a programming textbook, I'll spare you the dissertation on how the program operates (it's a lot easier to use the thing than to explain why it works). If you're interested please study the source code comments and feel free to experiment. After all, the worst thing you can do is crash the computer. I did it many times before I got Clock–Calendar to work (crashes in IRQ-driven routines are really exciting).

If after reading through the source code, you still aren't sure as to how the thing runs, please contact me and I'll send you a detailed description of Clock–Calendar and how it operates. I may be contacted via voice at (312) 595–3356 between the hours of 1930 and 2230 Central Time weekdays or all day on most Sundays.

Listing 1: Run this program to generate the clock display program "clk4864" on disk. See text for details.

```
EA 100 rem  generator for "clk4864"
MO 110 nd$="clk4864": rem  name of program
DH 120 nd=729: sa=4864: ch=65204
```

For lines 130 to 200, use the standard generator program on page 5

```
MP 1000 data  32,  32,  45,  32,  32,  45,  32,  32
AO 1010 data   0,  32,  32,  58,  32,  32,  58,  32
CN 1020 data  32,   0, 255, 255, 255,   0,   7,   0
AA 1030 data   0,   0,   0,   0, 101, 250, 255, 255
JO 1040 data 255,  76, 105,  19,  76,  42,  19,   0
MB 1050 data   0,   0, 174,  20,   3, 172,  21,   3
NL 1060 data 224, 206, 208,  52, 192,  19, 208,  48
IG 1070 data 120, 174,  28,  19, 172,  29,  19, 142
LI 1080 data  20,   3, 140,  21,   3, 174, 137,  21
JF 1090 data 172, 138,  21, 142,  38,   3, 140,  39
FF 1100 data   3, 174, 198,  21, 172, 199,  21, 142
FP 1110 data   0,  10, 140,   1,  10,  88, 160,   3
AE 1120 data 169,   0, 153,  24,  19, 136,  16, 250
MB 1130 data  96, 174,  20,   3, 172,  21,   3, 224
DI 1140 data 206, 208,   4, 192,  19, 240,  86, 142
NJ 1150 data  28,  19, 140,  29,  19, 120, 162, 206
MB 1160 data 160,  19, 142,  20,   3, 140,  21,   3
GM 1170 data 174,  38,   3, 172,  39,   3, 142, 137
JP 1180 data  21, 140, 138,  21, 162, 116, 160,  21
FM 1190 data 142,  38,   3, 140,  39,   3, 174,   0
KJ 1200 data  10, 172,   1,  10, 142, 198,  21, 140
GA 1210 data 199,  21, 162, 189, 160,  21, 142,   0
ML 1220 data  10, 140,   1,  10, 173,   8, 221, 141
DP 1230 data   8, 221,  88, 160,   3, 185, 212,  21
LE 1240 data 153,  24,  19, 136,  16, 247, 200, 140
OP 1250 data  21,  19, 140, 216,  21,  96, 216, 173
LN 1260 data   8, 221, 240,   8, 169,   0, 141, 216
ND 1270 data  21, 108,  28,  19,  44, 216,  21,  48
OM 1280 data 248, 206, 216,  21, 162,   3, 160,   0
DF 1290 data 189,   8, 221, 202,  48,   6, 153,  39
II 1300 data  19, 200, 208, 244, 173,  39,  19,  48
IB 1310 data   8, 201,  18, 208,  14, 169,   0, 240
DA 1320 data  10,  41, 127, 201,  18, 176,   4, 248
FE 1330 data 105,  18, 216, 141,  39,  19,  44,  32
NO 1340 data  19,  48,  38, 160,   1, 185,  30,  19
PH 1350 data 217,  39,  19, 208,  28, 136,  16, 245
DL 1360 data 169,  15, 141,  24, 212, 141,  14, 212
PH 1370 data 141,  15, 212, 162, 250, 160,  33, 142
```

```
IL 1380 data  20, 212, 140,  18, 212, 136, 140,  18
GF 1390 data 212, 160,   2, 185,  39,  19, 208,  73
NM 1400 data 136,  16, 248, 173,  20,  19,  32, 170
DL 1410 data  21, 162,  40,  74, 176,   4,  74, 176
PC 1420 data   1, 232, 142, 201,  21, 173,  18,  19
GI 1430 data  32, 170,  21, 170, 202, 173,  19,  19
IP 1440 data 221, 200,  21, 208,  30, 224,  11, 208
BK 1450 data  13, 173,  20,  19,  32, 183,  21, 141
LD 1460 data  20,  19, 169,   1, 208,   6, 173,  18
CN 1470 data  19,  32, 183,  21, 141,  18,  19, 169
LD 1480 data   1, 208,   3,  32, 183,  21, 141,  19
CO 1490 data  19, 173,  18,  19,  32, 148,  21, 201
HI 1500 data  48, 208,   2, 169,  32, 141,   0,  19
NA 1510 data 142,   1,  19, 173,  19,  19,  32, 148
KG 1520 data  21, 141,   3,  19, 142,   4,  19, 173
IH 1530 data  20,  19,  32, 148,  21, 141,   6,  19
FE 1540 data 142,   7,  19, 173,  39,  19,  32, 148
ED 1550 data  21, 141,   9,  19, 142,  10,  19, 173
LD 1560 data  40,  19,  32, 148,  21, 141,  12,  19
PB 1570 data 142,  13,  19, 173,  41,  19,  32, 148
GC 1580 data  21, 141,  15,  19, 142,  16,  19, 173
PC 1590 data  21,  19, 208, 101, 173,  22,  19, 160
JE 1600 data   7, 153,  32, 216, 153,  72, 216, 136
KI 1610 data  16, 247, 174,  23,  19, 160,   7, 185
KG 1620 data   9,  19, 224,   0, 240,   2,   9, 128
EH 1630 data 153,  32,   4, 185,   0,  19, 224,   0
IP 1640 data 240,   2,   9, 128, 153,  72,   4, 136
MK 1650 data  16, 229, 169,   0, 160,  72,  32, 106
OA 1660 data  21, 162,   9, 160,  19,  32,  68,  21
DH 1670 data 169,   0, 160, 152,  32, 106,  21, 162
FL 1680 data   0, 160,  19,  32,  68,  21, 169,   8
AN 1690 data 160,  72,  32, 106,  21,  32,  87,  21
NG 1700 data 169,   8, 160, 152,  32, 106,  21,  32
MN 1710 data  87,  21,  36, 215,  16,   3,  76,  51
DO 1720 data 255, 108,  28,  19, 142,  77,  21, 140
DO 1730 data  78,  21, 160,   0, 185,   0,   0, 240
LJ 1740 data  24,  32, 202, 205, 200, 208, 245, 160
LP 1750 data   8, 173,  22,  19, 174,  23,  19, 240
DO 1760 data   2,   9,  64,  32, 202, 205, 136, 208
DI 1770 data 240,  96, 162,  18,  32, 204, 205, 232
AD 1780 data 152,  76, 204, 205,  72,  36, 215,  16
NC 1790 data  14, 165, 154, 201,   3, 208,   8, 173
KB 1800 data  21,  19, 208,   3, 206,  21,  19, 104
IL 1810 data  32, 121, 239,  44,  21,  19,  16,   3
DF 1820 data 238,  21,  19,  96,  32, 160,  21,  72
CN 1830 data 138,   9,  48, 170, 104,   9,  48,  96
LG 1840 data  72,  41,  15, 170, 104,  74,  74,  74
PD 1850 data  74,  96,  32, 160,  21, 168, 138,  24
OH 1860 data 136,  48,   9, 105,  10, 208, 249, 248
FK 1870 data  24, 105,   1, 216,  96,  32, 105,  19
OC 1880 data 169,   0, 141,  34,  10,  76,   0,   0
FO 1890 data  49,  40,  49,  48,  49,  48,  49,  49
FL 1900 data  48,  49,  48,  49,  67,  76,  75,   0
MA 1910 data   0
```

Listing 2: This creates the clock/alarm set program "clk5632".

```
IP 100 rem  generator for "clk5632"
KN 110 nd$ = "clk5632": rem  name of program
NH 120 nd = 468: sa = 5632: ch = 49184
```

For lines 130 to 200, use the standard generator program on page 5

```
BN 1000 data  41,   1, 141, 208,  23, 165, 241,  72
LO 1010 data 173,   0, 255,  72, 162,  14, 142,   0
HC 1020 data 255, 232, 142,  24, 212, 169, 183, 141
HK 1030 data  60,   3,  32,  93,  23, 169, 147,  32
IB 1040 data 210, 255, 162,   2, 160,   0,  32,  60
LJ 1050 data  23, 173, 208,  23,  10, 170, 189, 113
FE 1060 data  23, 188, 114,  23,  32,  64,  23, 162
PL 1070 data   2,  56, 152, 233,   7, 168,  32,  60
JA 1080 data  23,  32,  80,  23, 133, 252, 120,  32
NE 1090 data 111, 205,  88,  32, 228, 255, 240, 251
IH 1100 data  72,  36, 215,  16,   8, 120,  32, 172
EJ 1110 data 205,  88,  76,  96,  22,  32, 159, 205
BL 1120 data 104, 164, 252, 240,   4, 201,  20, 240
OG 1130 data 185, 192,   4, 144,   6, 201,  13, 208
EI 1140 data 213, 240,  28, 201,  48, 144, 207, 201
DK 1150 data  58, 176, 203,  32, 210, 255, 153, 204
OB 1160 data  23, 230, 252, 192,   1, 208, 191, 169
FJ 1170 data  29,  32, 210, 255,  76,  70,  22, 174
PI 1180 data 204,  23, 173, 205,  23,  32,  48,  23
AG 1190 data 201,  36, 144,   6,  32,  85,  23,  76
OI 1200 data  34,  22, 141, 211,  23, 174, 206,  23
GL 1210 data 173, 207,  23,  32,  48,  23, 201,  96
DG 1220 data 176, 234, 141, 210,  23, 169,   0, 141
JD 1230 data 209,  23, 162,   4, 160,   0,  32,  60
FA 1240 data  23, 169, 175, 160,  23,  32,  64,  23
IG 1250 data  32,  93,  23,  32,  80,  23,  32, 228
HI 1260 data 255, 201,  78, 240,  73, 201,  89, 208
FB 1270 data 245, 173, 208,  23, 240,  18, 174, 211
NP 1280 data  23, 172, 210,  23, 142,  30,  19, 140
JI 1290 data  31,  19, 141,  32,  19,  76,  30,  23
MH 1300 data 173, 211,  23, 240,  15, 201,  18, 144
GE 1310 data  11, 248,  56, 233,  18, 216,   9, 128
OI 1320 data 141, 211,  23, 120, 173,  15, 221,  41
FB 1330 data 127, 141,  15, 221, 162,   2, 160,   3
LM 1340 data 189, 209,  23, 153,   8, 221, 202, 136
CL 1350 data 208, 246, 141,   8, 221,  88, 162, 173
OD 1360 data 160,   0, 142,  60,   3, 140,  24, 212
FJ 1370 data 104, 141,   0, 255, 104, 133, 241,  96
MA 1380 data  41,  15, 133, 252, 138,  10,  10,  10
PJ 1390 data  10,   5, 252,  96,  24,  76, 240, 255
JI 1400 data 133, 250, 132, 251, 160,   0, 177, 250
IF 1410 data 240,  10,  32, 210, 255, 200, 208, 246
BI 1420 data 169,   0, 133, 208,  96, 169,   6, 162
MO 1430 data 251, 160,  33, 208,   6, 169,  50, 162
DJ 1440 data 250, 160,  17, 141,   1, 212, 142,   6
PH 1450 data 212, 140,   4, 212, 136, 140,   4, 212
KP 1460 data  96, 117,  23, 147,  23, 158,  67,  79
MC 1470 data  82,  82,  69,  67,  84,  32,  84,  73
OD 1480 data  77,  69,  32,  40,  72,  72,  58,  77
MH 1490 data  77,  41,  58, 159,  32,  32,  32,  58
KE 1500 data  27,  81,   0, 158,  65,  76,  65,  82
PF 1510 data  77,  32,  84,  73,  77,  69,  32,  40
DM 1520 data  72,  72,  58,  77,  77,  41,  58, 159
AD 1530 data  32,  32,  32,  58,  27,  81,   0,   5
HF 1540 data  73,  83,  32,  84,  72,  73,  83,  32
OI 1550 data  84,  73,  77,  69,  32,  67,  79,  82
PH 1560 data  82,  69,  67,  84,  32,  40,  89,  47
EA 1570 data  78,  41,  63,   0,   0,   0,   0,   0
KG 1580 data   0,   0,   0,   0
```

Listing 3: Generator for "dt5632", the program to set the date.

```
IE 100 rem  generator for "dt5632"
FN 110 nd$ = "dt5632": rem  name of program
JF 120 nd = 414: sa = 5632: ch = 43027
```

For lines 130 to 200, use the standard generator program on page 5

```
PD 1000 data 165, 241,  72, 173,   0, 255,  72, 162
LO 1010 data  14, 142,   0, 255, 232, 142,  24, 212
IF 1020 data 169, 183, 141,  60,   3,  32,  38,  23
HK 1030 data 169, 147,  32, 210, 255, 162,   2, 160
AJ 1040 data   0,  32,  63,  23, 162,  95, 160,  23
FJ 1050 data  32,  67,  23, 162,   2, 160,  14,  32
AC 1060 data  63,  23,  32,  58,  23, 133, 252, 120
CL 1070 data  32, 111, 205,  88,  32, 228, 255, 240
CH 1080 data 251,  72,  36, 215,  16,   8, 120,  32
KH 1090 data 172, 205,  88,  76,  81,  22,  32, 159
AK 1100 data 205, 104, 164, 252, 240,   4, 201,  20
CF 1110 data 240, 195, 192,   6, 144,   6, 201,  13
MG 1120 data 208, 213, 240,  32, 201,  48, 144, 207
AJ 1130 data 201,  58, 176, 203,  32, 210, 255, 153
DH 1140 data 149,  23, 230, 252, 192,   1, 240,   4
OK 1150 data 192,   3, 208, 187, 169,  29,  32, 210
BD 1160 data 255,  76,  55,  22, 162,   2, 160,   5
NA 1170 data 185, 149,  23,  41,  15, 133, 252, 136
AB 1180 data 185, 149,  23,  10,  10,  10,  10,   5
BO 1190 data 252, 157, 155,  23, 136, 202,  16, 232
HM 1200 data  72, 173, 157,  23,  32,  12,  23, 162
GH 1210 data  41,  74, 176,   5,  74, 176,   2, 162
IA 1220 data  48, 142,  84,  23, 104, 208,   6,  32
CP 1230 data  30,  23,  76,  29,  22, 201,  19, 176
MC 1240 data 246,  32,  12,  23, 170, 202, 173, 156
KG 1250 data  23, 240, 236, 221,  83,  23, 176, 231
EF 1260 data 162,   4, 160,   0,  32,  63,  23, 162
CC 1270 data 120, 160,  23,  32,  67,  23,  32,  38
KA 1280 data  23,  32,  58,  23,  32, 228, 255, 201
FJ 1290 data  78, 240,  15, 201,  89, 208, 245, 162
AA 1300 data   2, 189, 155,  23, 157,  18,  19, 202
GH 1310 data  16, 247, 162, 173, 160,   0, 142,  60
AN 1320 data   3, 140,  24, 212, 104, 141,   0, 255
DF 1330 data 104, 133, 241,  96,  72,  74,  74,  74
OD 1340 data  74, 170, 104,  41,  15,  24, 202,  48
LH 1350 data   4, 105,  10, 208, 249,  96, 169,   6
GK 1360 data 162, 251, 160,  33, 208,   6, 169,  50
GO 1370 data 162, 250, 160,  17, 141,   1, 212, 142
DF 1380 data   6, 212, 140,   4, 212, 136, 140,   4
EK 1390 data 212,  96, 169,   0, 133, 208,  96,  24
CK 1400 data  76, 240, 255, 134, 250, 132, 251, 160
OK 1410 data   0, 177, 250, 240, 241,  32, 210, 255
JF 1420 data 200, 208, 246,  50,  41,  50,  49,  50
MD 1430 data  49,  50,  50,  49,  50,  49,  50, 158
LD 1440 data  84,  79,  68,  65,  89,  39,  83,  32
FK 1450 data  68,  65,  84,  69,  58, 159,  32,  32
CK 1460 data  32,  45,  32,  32,  45,  27,  81,   0
CO 1470 data   5,  73,  83,  32,  84,  72,  73,  83
KE 1480 data  32,  68,  65,  84,  69,  32,  67,  79
OD 1490 data  82,  82,  69,  67,  84,  32,  40,  89
BA 1500 data  47,  78,  41,  63,   0,   0,   0,   0
MO 1510 data   0,   0,   0,   0,   0,   0
```

Listing 4: CBM-format assembler source code for "clk4864".

```
;put"@0:clock-cal.src
.opt nos
;* * * * * * * * * * * * * * * * * * * * * * *
;*                                             *
;* clock & calendar display. . .               *
;c-128 mode, 40 or 80 columns                  *
;* with audible alarm function                 *
;* written 11-16-85  w.j. brier                *
;* revised  6-05-87                            *
;* copyright 1986                              *
;*                                             *
;* this program is not to be. . .              *
;* sold. it is permissible. . .                *
;* to copy it but credit must. . .             *
;* be given in the documentation               *
;*                                             *
;* see the documentation for. . .              *
;* instructions on using this. . .             *
;* program with your software. . .             *
;* * * * * * * * * * * * * * * * * * * * * * *

;* <<< program assignments >>> *
        ;
;system vectors & pointers. . .
dflto   =   $9a         ;output device
mode    =   $d7         ;display mode flag
cinv    =   $0314       ;normal irq vector
ibsout  =   $0326       ;normal chrout vector
system  =   $0a00       ;basic reset vector
rptflg  =   $0a22       ;keyb'd repeat flag
basrst  =   $4003       ;basic warm reset
tod2    =   $dd08       ;time of day clock #2
        ;
.pag
;screen editor functions. . .
wvdcm   =   $cdca       ;write to 8563 ram
wvdcr   =   $cdcc       ;write to 8563 register
        ;
;kernal functions. . .
chrout  =   $ef79       ;output a byte
irq     =   $fa65       ;normal irq
crti    =   $ff33       ;irq handler exit
        ;
;40 col vic screen ram. . .
vcram   =   $0400       ;start of display ram
vlram   =   $d800       ;start of color ram
        ;
;sid chip registers (voice 3). . .
frelo3  =   $d40e       ;frequency ctrl (lo)
frehi3  =   $d40f       ;frequency ctrl (hi)
vcreg3  =   $d412       ;control register
surel3  =   $d414       ;sustain/release
sigvol  =   $d418       ;sid volume
        ;
;80 col 8563 vdc assignments. . .
scram1  =   72          ;screen ram (time)
scram2  =   152         ;screen ram (date)
atram1  =   2120        ;attribute ram (time)
atram2  =   2200        ;attribute ram (date)
        ;
upreg   =   18          ;update register
        ;
*=$1300                 ;4864
        ;
.pag
;=============================
        ;
;user-accessible memory. . .
        ;
tdta    .byt '  -  -  ',0 ;ascii date
toda    .byt '  :  :  ',0 ;ascii time
tdtc    .byt 255,255,255 ;bcd date
dflg    .byt 0          ;1 = no display
cflg    .byt 7          ;display color
rflg    .byt 0          ;1 = reverse video
key     .byt 0,0,0,0
        ;
;exit vector. . .
irqa    .wor irq
        ;
;alarm registers. . .
altc    .byt 255,255    ;time (h:m)
```

```
aflg    .byt 255            ;enable flag
;-----------------------------------
;entry point to start display
clkon   jmp  start          ;start display
;-----------------------------------
;entry point to stop display
clkof   jmp  stop           ;stop display
;-----------------------------------
        ;
;bcd time storage
todc    .byt 0,0,0          ;hrs:min:sec
        ;
;==============================
        ;
;stop clock-calendar display
stop    ldx  cinv           ;irq vector
        ldy  cinv+1
        cpx  #<dcc          ;display vector
        bne  clkof2         ;not running
        cpy  #>dcc
        bne  clkof2         ;not running
        sei                 ;interrupts off
        ldx  irqa           ;original irq vector
        ldy  irqa+1
        stx  cinv
        sty  cinv+1
        ldx  bsouta
        ldy  bsouta+1
        stx  ibsout         ;restore chrout vector
        sty  ibsout+1
        ldx  alrst1
        ldy  alrst1+1
        stx  system         ;restore system vector
        sty  system+1
        cli                 ;interrupts on
        ldy  #3             ;offset
        lda  #0
clkof1  sta  key,y          ;wipe out id key
        dey
        bpl  clkof1
;loop
clkof2  rts
        ;
;-----------------------------------
;start clock-calendar display
start   ldx  cinv           ;irq vector
        ldy  cinv+1
        cpx  #<dcc
        bne  start1         ;change vector
        cpy  #>dcc
        beq  start3         ;display is runnning
start1  stx  irqa           ;save existing. . .
        sty  irqa+1         ;irq vector
        sei
        ldx  #<dcc          ;change. . .
        ldy  #>dcc          ;irq vector so that. . .
        stx  cinv           ;clock-calendar runs
        sty  cinv+1
        ldx  ibsout         ;save current. . .
        ldy  ibsout+1       ;chrout vector
        stx  bsouta
        sty  bsouta+1
        ldx  #<crdy         ;change. . .
        ldy  #>crdy         ;chrout vector so. . .
        stx  ibsout         ;delay routine. . .
        sty  ibsout+1       ;intercepts it
        ldx  system         ;basic reset
        ldy  system+1
        stx  alrst1         ;store
        sty  alrst1+1
        ldx  #<alrst        ;alternate reset
        ldy  #>alrst
        stx  system         ;new reset vector
        sty  system+1
        lda  tod2           ;give clock a kick. . .
        sta  tod2           ;to get it started
        cli
        ldy  #3             ;offset
start2  lda  keystr,y       ;key
        sta  key,y          ;enable key
        dey
        bpl  start2         ;loop
        iny
        sty  dflg           ;clear display flag
        sty  lflg           ;clear lockout flag
start3  rts
        ;
;==============================
;update & display time & date
dcc     cld                 ;binary mode
        lda  tod2           ;tenths of seconds
        beq  dcc02          ;tenths are zero
        lda  #0
        sta  lflg           ;clear lockout flag
dcc01   jmp  (irqa)         ;normal irq
dcc02   bit  lflg
        bmi  dcc01          ;update locked out
        dec  lflg           ;set lockout flag
        ;
;read clock registers. . .
        ldx  #3             ;tod2 offset
        ldy  #0             ;storage offset
dcc03   lda  tod2,x         ;fetch time value
        dex
        bmi  dcc04          ;finished
        sta  todc,y         ;save in buffer
        iny
        bne  dcc03          ;loop
        ;
;test am/pm flag & adjust hours
dcc04   lda  todc           ;bcd hours
        bmi  dcc05          ;it's pm
        cmp  #$12
        bne  dcc06          ;not midnite
        lda  #0
        beq  dcc06          ;set hours to midnite
dcc05   and  #%01111111     ;mask am/pm bit
        cmp  #$12
        bcs  dcc06          ;it's noon straight up
        sed                 ;decimal mode
        adc  #$12           ;change to 24 hours
        cld                 ;binary mode
dcc06   sta  todc           ;converted hours
        ;
;test for alarm time. . .
        bit  aflg           ;check flag
        bmi  dcc09          ;alarm not enabled
        ldy  #1             ;offset
dcc07   lda  altc,y         ;alarm time
        cmp  todc,y         ;check against tod
        bne  dcc09          ;not time
        dey
        bpl  dcc07          ;loop
        ;
;sound alarm. . .
        lda  #15
        sta  sigvol         ;max volume
        sta  frelo3         ;set frequency
        sta  frehi3
        ldx  #250           ;duration
        ldy  #33            ;sawtooth waveform
        stx  surel3         ;sustain/release
        sty  vcreg3         ;attack/decay
        dey
        sty  vcreg3         ;gate voice
        ;
;test for stroke of midnite. . .
dcc09   ldy  #2             ;todc offset
        ;
dcc10   lda  todc,y         ;fetch time value
        bne  dcc16          ;not midnite
        dey
        bpl  dcc10          ;loop
        ;
;test for leap year. . .
        lda  tdtc+2         ;current year
        jsr  bcdof          ;change to binary
        ldx  #$28
        lsr  a              ;divide by 2
        bcs  dcc11          ;odd year
        lsr  a              ;divide again
        bcs  dcc11          ;non-leap year
        inx                 ;leap year
dcc11   stx  clut+1         ;last day in feb.
        ;
;adjust date for end of month. . .
        lda  tdtc           ;fetch month
        jsr  bcdof          ;change to binary
        tax
        dex                 ;calendar lookup offset
        lda  tdtc+1         ;current day
        cmp  clut,x         ;last day of month
        bne  dcc14          ;not end of month
        cpx  #11
        bne  dcc12          ;not december
        lda  tdtc+2         ;current year
        jsr  idv            ;bump year
        sta  tdtc+2         ;save new year
        lda  #1
        bne  dcc13          ;set month to january
dcc12   lda  tdtc           ;current month
        jsr  idv            ;bump
dcc13   sta  tdtc           ;new month
        lda  #1
        bne  dcc15          ;set day to 1st of month
dcc14   jsr  idv            ;bump day
dcc15   sta  tdtc+1         ;new day
        ;
;decode date for display. . .
dcc16   lda  tdtc           ;current month
        jsr  bcdasc         ;decode to ascii
        cmp  #'0
        bne  dcc17
        lda  #32            ;blank leading zero
dcc17   sta  tdta           ;save tens
        stx  tdta+1         ;save units
        lda  tdtc+1         ;current day
        jsr  bcdasc
        sta  tdta+3
        stx  tdta+4
        lda  tdtc+2         ;current year
        jsr  bcdasc
        sta  tdta+6
        stx  tdta+7
        ;
;decode time for display. . .
        lda  todc           ;hours
        jsr  bcdasc
        sta  toda
        stx  toda+1
        ;
        lda  todc+1         ;minutes
        jsr  bcdasc
        sta  toda+3
        stx  toda+4
        ;
        lda  todc+2         ;seconds
        jsr  bcdasc
        sta  toda+6
        stx  toda+7
        ;
;display time & date. . .
        lda  dflg
        bne  dcc23          ;display inhibited
        ;
;display on 40 column screen. . .
dcc18   lda  cflg           ;display color
        ldy  #7             ;offset
dcc19   sta  vlram+32,y     ;color
        sta  vlram+72,y
        dey
        bpl  dcc19          ;loop
        ldx  rflg           ;normal/reverse flag
        ldy  #7             ;offset
dcc20   lda  toda,y         ;fetch time
        cpx  #0             ;test reverse flag
        beq  dcc21          ;not reversed
        ora  #128           ;reverse
dcc21   sta  vcram+32,y
        lda  tdta,y         ;fetch date
        cpx  #0
        beq  dcc22
        ora  #128
dcc22   sta  vcram+72,y
        dey
        bpl  dcc20
        ;
;display on 80 column screen. . .
        lda  #>scram1       ;address for. . .
        ldy  #<scram1       ;time display
        jsr  setram         ;set up vdc ram
        ldx  #<toda         ;petascii time
```

```
        ldy  #>toda
        jsr  dtod       ;display
        lda  #>scram2   ;address for. . .
        ldy  #<scram2   ;date display
        jsr  setram
        ldx  #<tdta     ;petascii date
        ldy  #>tdta
        jsr  dtod       ;display
        ;
;set 80 column attributes. . .
        lda  #>atram1   ;address for. . .
        ldy  #<atram1   ;time attribute
        jsr  setram
        jsr  atrst      ;set up attributes
        lda  #>atram2   ;address for. . .
        ldy  #<atram2   ;date attribute
        jsr  setram
        jsr  atrst      ;set up attributes
        bit  mode
        bpl  dcc23      ;40 column mode
        jmp  crti       ;bypass rest of irq
dcc23   jmp  (irqa)     ;continue irq
        ;
;==============================
;display time or date
dtod    stx  dtod02     ;source address
        sty  dtod02+1
        ldy  #0         ;offset
dtod01  .byt $b9        ;lda llhh,y op-code
dtod02  *=*+2           ;source address
        beq  atrst3     ;end of string
        jsr  wvdcm      ;output to 8563
        iny
        bne  dtod01     ;loop
        ;
;--------------------------------
;set up display attributes
atrst   ldy  #8         ;counter
atrst1  lda  cflg       ;get color
        ldx  rflg
        beq  atrst2     ;no reverse display
        ora  #%01000000 ;reverse
atrst2  jsr  wvdcm      ;output to 8563
        dey
        bne  atrst1     ;loop
atrst3  rts
        ;
;--------------------------------
;set up vdc ram address
setram  ldx  #upreg     ;update register
        jsr  wvdcr      ;write
        inx
        tya             ;swap hi byte
        jmp  wvdcr
        ;
;--------------------------------
;chrout intercept & trap
crdy    pha             ;save printing char.
        bit  mode
        bpl  crdy01     ;in 40 columns
        lda  dflto      ;output device
        cmp #3
        bne  crdy01     ;not screen
        lda  dflg       ;test display flag
        bne  crdy01     ;display inhibited
        dec  dflg       ;block display
crdy01  pla             ;recover character
        .byt 32         ;jsr op-code
bsouta  .wor chrout     ;normal chrout
        bit  dflg
        bpl  crdy02     ;no reset needed
        inc  dflg       ;clear for display
crdy02  rts
        ;
;--------------------------------
;convert bcd to petascii
bcdasc  jsr  bcdbin     ;bcd to binary
        pha             ;save tens value
        txa             ;units value
        ora  #48        ;change to petascii
        tax             ;hold
        pla             ;fetch tens
        ora  #48
        rts
        ;
;--------------------------------
;bcd digit to 2 binary digits
bcdbin  pha             ;save bcd value
        and  #15        ;mask tens nybble
        tax             ;units in .x register
        pla             ;fetch bcd value
        lsr  a          ;shift tens nybble
        lsr  a
        lsr  a
        lsr  a
        rts
        ;
;--------------------------------
;bcd digit to 1 binary digit
bcdof   jsr  bcdbin     ;bcd to binary
        tay             ;tens value
        txa             ;units
        clc
bcdof1  dey             ;counter
        bmi  idv01
;finished
        adc  #10
        bne  bcdof1
        ;
;--------------------------------
;increment date value
idv     sed             ;decimal mode
        clc
        adc  #1
        cld             ;binary mode
idv01   rts
        ;
;--------------------------------
;alternate basic reset
alrst   jsr  start      ;restart clock
        lda  #0
        sta  rptflg     ;repeat off
        .byt $4c        ;jmp llhh op-code
alrst1  *=*+2           ;reset jump
        ;
;--------------------------------
;calendar lookup table
clut    .byt $31        ;jan
        .byt $28        ;feb
        .byt $31        ;mar
        .byt $30        ;apr
        .byt $31        ;may
        .byt $30        ;jun
        .byt $31        ;jul
        .byt $31        ;aug
        .byt $30        ;sep
        .byt $31        ;oct
        .byt $30        ;nov
        .byt $31        ;dec
        ;
;--------------------------------
;program storage
keystr  .byt 'clk',0
lflg    *=*+1           ;display lockout
        ;
;==============================
.end
```

Listing 5: Assembler source code for "clk5632"

```
;put"@0:clockset.src
.opt nos
;**************************
;*                        *
;* clock & alarm setup    *
;* written 12-02-85 w.j. brier *
;* revised  1-18-87       *
;* copyright (c) 1985     *
;* all rights reserved    *
;*                        *
;* use with clock-calendar 128 *
;*                        *
;* set time of day: sys 5632,0 *
;* set alarm time: sys 5632,1 *
;*                        *
;* enter time in 24 hour format *
;**************************
        ;
;* <<< program assignments >>> *
        ;
;system vectors & pointers. . .
        ;
ndx     =  $d0          ;keyboard queue
mode    =  $d7          ;40/80 column mode
color   =  $f1          ;next attribute
keychk  =  $033c        ;key decode vector
        ;
curon   =  $cd6f        ;flash cursor
curof1  =  $cd9f        ;kill cursor (40 col)
curof2  =  $cdac        ;kill cursor (80 col)
        ;
sigvol  =  $d418        ;volume control
tod2    =  $dd08        ;time of day clock #2
mmu     =  $ff00        ;memory configuration
        ;
chrout  =  $ffd2        ;output a byte
getin   =  $ffe4        ;get a byte
plot    =  $fff0        ;position cursor
        ;
;miscellaneous pointers. . .
ptr     =  $fa          ;zero page pointer
ctr     =  $fc          ;counter
        ;
;clock-calendar locations. . .
altc    =  $131e        ;alarm register
aflg    =  $1320        ;alarm enable flag
        ;
;video constants. . .
wht     =  5            ;white text
cr      =  13           ;carriage return
del     =  20           ;delete
esc     =  27           ;escape character
right   =  29           ;cursor right
clr     =  147          ;clear screen
yel     =  158          ;yellow text
cyn     =  159          ;cyan text
        ;
*=$1600                 ;5632
        ;
;################
;#              #
;#  c-128 time/alarm setup  #
;#              #
;################
        ;
;initial setup. . .
stim    and  #1         ;mask garbage &. . .
        sta  sflg       ;set entry mode flag
        lda  color      ;current attribute
        pha             ;save
        lda  mmu        ;configuration
        pha             ;save on stack
        ldx  #14
        stx  mmu        ;enable kernal
        inx
        stx  sigvol     ;maximum volume
        lda  #183
        sta  keychk     ;bypass f keys
        jsr  chime      ;signal user
        lda  #clr
        jsr  chrout     ;clear screen
        ;
;display input prompt. . .
stim01  ldx  #2         ;row
        ldy  #0         ;column
        jsr  plota      ;position cursor
        lda  sflg       ;entry mode
        asl  a          ;double
        tax             ;becomes prompt offset
        lda  ptab,x     ;prompt address
        ldy  ptab+1,x
        jsr  prnt       ;output
        ldx  #2         ;row
        sec
        tya
        sbc  #7         ;generate column value
        tay
        jsr  plota      ;position cursor
        jsr  clrq       ;clear keyb'd queue
        sta  ctr        ;clear input counter
        ;
;fetch user input. . .
stim02  sei             ;interrupts off
        jsr  curon      ;flash cursor
```

```
        cli                 ;interrupts on
        ;
stim03  jsr  getin          ;fetch keypress
        beq  stim03         ;no input
        pha                 ;save keypress
        bit  mode
        bpl  stim04         ;40 columns
        sei
        jsr  curof2         ;kill 80 col cursor
        cli
        jmp  stim05
stim04  jsr  curof1         ;kill 40 col cursor
        ;
;filter & store input. . .
stim05  pla                 ;retrieve keypress
        ldy  ctr            ;fetch count
        beq  stim06
        cmp #del
        beq  stim01         ;input deleted
stim06  cpy  #4
        bcc  stim07         ;more input needed
        cmp #cr
        bne  stim02         ;not <return>
        beq  stim08         ;finished
        ;
stim07  cmp #'0
        bcc  stim02         ;out of range
        cmp #':
        bcs  stim02         ;out of range
        jsr  chrout         ;echo digit
        sta  buf,y          ;store
        inc  ctr            ;bump character count
        cpy  #1
        bne  stim02         ;loop
        lda  #right
        jsr  chrout         ;jump over colon
        jmp  stim02         ;loop
        ;
;encode time into bcd. . .
stim08  ldx  buf            ;fetch hours (tens)
        lda  buf+1          ;fetch hours (units)
        jsr  ascbcd         ;convert
        cmp #$24
        bcc  stim10
stim09  jsr  buzzer         ;illegal value
        jmp  stim01         ;reenter
stim10  sta  todc+2         ;save
        ldx  buf+2          ;fetch minutes (tens)
        lda  buf+3          ;fetch minutes (units)
        jsr  ascbcd         ;convert
        cmp #$60
        bcs  stim09         ;illegal minute value
        sta  todc+1         ;save
        lda  #0
        sta  todc           ;zero seconds
        ;
;confirm time entry. . .
        ldx  #4
        ldy  #0
        jsr  plota
        lda  #<timp3        ;'correct?'
        ldy  #>timp3
        jsr  prnt
        jsr  chime
        jsr  clrq
stim11  jsr  getin          ;fetch keypress
        cmp #'n
        beq  stim15         ;abort
        cmp #'y
        bne  stim11         ;loop
        lda  sflg
        beq  stim12         ;setting tod
        ;
;set alarm time & enable flag. . .
        ldx  todc+2         ;entered hours
        ldy  todc+1         ;entered minutes
        stx  altc           ;set alarm register
        sty  altc+1
        sta  aflg           ;set alarm flag
        jmp  stim15         ;exit
        ;
;set time of day. . .
stim12  lda  todc+2         ;hours
        beq  stim13         ;is midnite
        cmp #$12
        bcc  stim13         ;is am
        ;
;convert 24 hour entry to 12 hour. . .
        sed                 ;decimal mode
        sec
        sbc  #$12           ;change to 12 hour time
        cld                 ;binary mode
        ora  #128           ;set pm bit
        sta  todc+2         ;save hours
        ;
;set clock registers. . .
        sei
stim13  lda  tod2+7         ;control register
        and  #127           ;setting time of day
        sta  tod2+7
        ldx  #2             ;todc offset
        ldy  #3             ;tod2 offset
stim14  lda  todc,x         ;fetch time &. . .
        sta  tod2,y         ;store in cia #2. . .
        dex                 ;hardware registers
        dey
        bne  stim14         ;loop
        sta  tod2           ;start clock
        cli
        ;
;restore system & exit. . .
stim15  ldx  #173
        ldy  #0
        stx  keychk         ;enable f keys
        sty  sigvol         ;sound off
        pla
        sta  mmu            ;restore configuration
        pla
        sta  color          ;restore attribute
        rts
        ;
;============================
;convert ascii to bcd
ascbcd  and  #15            ;units to binary
        sta  ctr            ;store
        txa                 ;swap tens
        asl  a              ;shift tens to. . .
        asl  a              ;high nybble
        asl  a
        asl  a
        ora  ctr            ;combine nybbles
        rts
        ;
;----------------------------
;position cursor
plota   clc
        jmp  plot
        ;
;----------------------------
;print to screen
prnt    sta  ptr            ;save text pointer
        sty  ptr+1
        ldy  #0             ;offset
prnt01  lda  (ptr),y        ;fetch
        beq  clrq01         ;exit
        jsr  chrout         ;print
        iny
        bne  prnt01         ;loop
        ;
;----------------------------
;clear keyboard queue
clrq    lda  #0
        sta  ndx
clrq01  rts
        ;
;----------------------------
;buzzer
buzzer  lda  #6
        ldx  #251
        ldy  #33
        bne  signal
        ;
;----------------------------
;chime
chime   lda  #50
        ldx  #250
        ldy  #17
        ;
;----------------------------
;signal user
signal  sta  54273
        stx  54278
        sty  54276
        dey
        sty  54276
        rts
        ;
;----------------------------
;input prompt look-up table
ptab    .wor timp1,timp2
        ;
;----------------------------
;input prompts
timp1   .byt yel
        .byt 'correct time (hh:mm):'
        .byt cyn,'[3 spcs]:',esc,'q',0
timp2   .byt yel
        .byt 'alarm time (hh:mm):'
        .byt cyn,'[3 spcs]:',esc,'q',0
timp3   .byt wht
        .byt 'is this time cor'
        .byt 'rect (y/n)?',0
;----------------------------
        ;
;program storage
buf     *=*+4               ;input buffer
sflg    *=*+1               ;entry mode
todc    *=*+3               ;bcd time (s:m:h)
;
;============================
.end
```

Listing 6: Source code for date-set program "dt5632"

```
;put"@0:dateset.src
.opt nos
;**************************
;*                        *
;* calendar date setup    *
;* written 11-29-85 w.j. brier *
;* revised 1-18-87        *
;* copyright (c) 1985     *
;* all rights reserved    *
;*                        *
;* use with clock-calendar 128 *
;* set date: sys 5632     *
;* enter date as: mm-dd-yy *
;**************************
        ;
;*   program assignments   *
        ;
;system vectors & pointers. . .
ndx     =   $d0             ;keyboard queue
mode    =   $d7             ;40/80 column mode
color   =   $f1             ;next attribute
keychk  =   $033c           ;key decode vector
curon   =   $cd6f           ;flash cursor
curof1  =   $cd9f           ;kill cursor (40 col)
curof2  =   $cdac           ;kill cursor (80 col)
sigvol  =   $d418           ;volume control
mmu     =   $ff00           ;memory management
chrout  =   $ffd2           ;output a byte
getin   =   $ffe4           ;get a byte
plot    =   $fff0           ;position cursor
        ;
;miscellaneous pointers. . .
ptr     =   $fa             ;zero page pointer
ctr     =   $fc             ;counter
        ;
;clock-calendar location. . .
tdtc    =   $1312           ;bcd date in clock
        ;
;video constants. . .
wht     =   5               ;white text
cr      =   13              ;carriage return
del     =   20              ;delete
esc     =   27              ;escape character
right   =   29              ;cursor right
clr     =   147             ;clear screen
yel     =   158             ;yellow text
cyn     =   159             ;cyan text
        ;
*=$1600                     ;5632
        ;
;################
```

```
;#                                #
;# c-128 calendar date setup #
;#                                #
;################
        ;
;initial setup. . .
date    lda  color      ;current attribute
        pha             ;save
        lda  mmu        ;configuration
        pha             ;save on stack
        ldx  #14
        stx  mmu        ;enable kernal
        inx
        stx  sigvol     ;maximum volume
        lda  #183
        sta  keychk     ;bypass f keys
        jsr  chime      ;signal user
        lda  #clr
        jsr  chrout     ;clear screen
        ;
;display input prompt. . .
date01  ldx  #2         ;row
        ldy  #0         ;column
        jsr  plota      ;position cursor
        ldx  #<datep1   ;'today's date'
        ldy  #>datep1
        jsr  prnt       ;display prompt
        ldx  #2         ;position cursor. . .
        ldy  #14        ;to accept. . .
        jsr  plota      ;user input
        jsr  clrq       ;clear keyb'd queue
        sta  ctr        ;clear input counter
        ;
;fetch user input. . .
date02  sei             ;interrupts off
        jsr  curon      ;flash cursor
        cli             ;interrupts on
date03  jsr  getin      ;fetch keypress
        beq  date03     ;no input
        pha             ;save keypress
        bit  mode
        bpl  date04     ;40 columns
        sei
        jsr  curof2     ;kill 80 col cursor
        cli
        jmp  date05
date04  jsr  curof1     ;kill 40 col cursor
        ;
;filter & store input. . .
date05  pla             ;retrieve keypress
        ldy  ctr        ;fetch character count
        beq  date06
        cmp #del
        beq  date01     ;deleted
date06  cpy  #6
        bcc  date07     ;more input needed
        cmp #cr
        bne  date02     ;not <return>
        beq  date09     ;end of input
date07  cmp #'0
        bcc  date02     ;out of range
        cmp #':
        bcs  date02     ;out of range
        jsr  chrout     ;echo character
        sta  buf,y      ;store
        inc  ctr
        cpy  #1
        beq  date08
        cpy  #3
        bne  date02     ;loop
date08  lda  #right
        jsr  chrout     ;jump over hyphen
        jmp  date02     ;loop
        ;
;encode date into bcd. . .
date09  ldx  #2         ;bcd offset
        ldy  #5         ;ascii offset
date10  lda  buf,y      ;fetch units
        and  #15        ;mask hi nybble
        sta  ctr        ;store
        dey
        lda  buf,y      ;fetch tens
        asl  a          ;shift lo nybble. . .
        asl  a          ;to high
        asl  a
        asl  a
        ora  ctr        ;combine w/units
        sta  dbuf,x     ;save bcd digit
        dey
        dex
        bpl  date10     ;loop
        ;
;test for leap year entry. . .
        pha             ;save month value
        lda  dbuf+2     ;year entry
        jsr  bcbin      ;change to binary
        ldx  #$29       ;non-leap year test value
        lsr  a          ;divide year by 2
        bcs  date11     ;non-leap year
        lsr  a
        bcs  date11
        ldx  #$30       ;bump test value
date11  stx  clut+1     ;test value
        ;
;check for valid input month. . .
        pla             ;input month
        bne  date13     ;not zero month
date12  jsr  buzzer     ;error
        jmp  date01     ;reenter
date13  cmp #$13
        bcs  date12     ;too high
        ;
;check for valid input day. . .
        jsr  bcbin      ;change to binary
        tax
        dex             ;make table offset
        lda  dbuf+1     ;entry day
        beq  date12     ;zero for day
        cmp clut,x      ;table
        bcs  date12     ;too high for month
        ;
;confirm entry. . .
        ldx  #4
        ldy  #0
        jsr  plota
        ldx  #<datep2   ;'is this correct?'
        ldy  #>datep2
        jsr  prnt
        jsr  chime
        jsr  clrq
date14  jsr  getin      ;fetch keypress
        cmp #'n
        beq  date16     ;abort
        cmp #'y
        bne  date14     ;loop
        ;
;transfer date to calendar storage. . .
        ldx  #2         ;offset
date15  lda  dbuf,x
        sta  tdtc,x     ;store in calendar
        dex
        bpl  date15     ;loop
        ;
;restore system & exit. . .
date16  ldx  #173
        ldy  #0
        stx  keychk     ;enable f keys
        sty  sigvol     ;shut off sid
        pla
        sta  mmu        ;restore configuration
        pla
        sta  color      ;restore attribute
        rts
        ;
;==============================
;change bcd digit to binary digit
bcbin   pha             ;save bcd digit
        lsr  a          ;shift tens nybble. . .
        lsr  a          ;to lo position
        lsr  a
        lsr  a
        tax             ;becomes tens counter
        pla             ;fetch bcd digit
        and  #15        ;mask tens nybble
        clc
bcbin1  dex             ;step tens counter
        bmi  bcbin2     ;finished
        adc  #10        ;add 10 to units
        bne  bcbin1     ;loop
bcbin2  rts
        ;
;-------------------------------
;buzzer tone
buzzer  lda  #6         ;frequency
        ldx  #251       ;duration
        ldy  #33        ;sawtooth
        bne  signal
        ;
;-------------------------------
;chime tone
chime   lda  #50
        ldx  #250
        ldy  #17        ;triangle
        ;
;-------------------------------
;signal user
signal  sta  54273
        stx  54278
        sty  54276
        dey
        sty  54276      ;gate tone on
        rts
        ;
;-------------------------------
;clear keyboard queue
clrq    lda  #0
        sta  ndx
clrq01  rts
        ;
;-------------------------------
;position cursor
plota   clc
        jmp  plot
        ;
;-------------------------------
;print to screen
prnt    stx  ptr        ;string address
        sty  ptr+1
        ldy  #0         ;offset
prnt01  lda  (ptr),y    ;fetch
        beq  clrq01
        jsr  chrout
        iny
        bne  prnt01     ;loop
        ;
;-------------------------------
;calendar lookup table
clut    .byt $32        ;jan+1
        .byt $29        ;feb+1
        .byt $32        ;mar+1
        .byt $31        ;apr+1
        .byt $32        ;may+1
        .byt $31        ;jun+1
        .byt $32        ;jul+1
        .byt $32        ;aug+1
        .byt $31        ;sep+1
        .byt $32        ;oct+1
        .byt $31        ;nov+1
        .byt $32        ;dec+1
        ;
;-------------------------------
;user prompts
datep1  .byt yel,'today''s date:'
        .byt cyn,'[3 spcs]-[2 spcs]-'
        .byt esc,'q',0
        ;
datep2  .byt wht,'is this date'
        .byt '  correct (y/n)?',0
        ;
;-------------------------------
;program storage
buf     *=*+6           ;input buffer
dbuf    *=*+3           ;date encoding buffer
        ;
;==============================
.end
```

Amiga Dispatches

by Tim Grantham, Toronto, Ontario

This edition of Amiga Dispatches will be the last to appear in the original Transactor. I thought it fitting then, that this column cover a subject I hope will soon be very important to many of you – moving up from programming the Commodore 64/128 to programming on the Amiga.

The idea for this column came from an electronic letter I received from Tom Brown on PeopleLink. Tom wrote:

> *I am sure I am not alone in my confusion, trying to leap the gap from a 128/64 to the Amiga. Perhaps you could give some advice to the flood of new Amiga users:*
>
> *1) What is necessary for the beginning Amiga programmer to learn (assuming a background in Basic and 6502 assembler)?*
> *2) Where does one get that info i.e. a suggested reading list?*
> *3) What files should a newcomer look for in the public domain for programming, disk doctoring, and so on?*
> *4) Which programming language? Look for a Basic compiler or go the route with C (or even ML)?. Which C compilers to use without breaking the budget?*

As I have discussed in previous columns, the journey from the land of 8-bit machines to the Olympian heights inhabited by the Amiga is not an easy one. For me, and I suspect for most others, it meant not only learning an entirely different operating system but learning C as well. (The fruits of my efforts can be found in **Keep v1.1**, which I have posted to all the major information networks and which will be available on the next Transactor Amiga disk.) For this edition of AD, I approached a number of people who have climbed that steep learning curve and asked them to suggest ways to make it less arduous and more enjoyable. I suggested that they target their answers to the person I consider to be a typical Transactor reader – one who is fairly well versed in Commodore BASIC; who has some acquaintance, at least, with 6502 assembly language; who is prepared to spend some time and money on books and tools; but who has little experience or knowledge of the more complex operating systems found on multi-user computers and on the Amiga.

Many people consider C to be the 'natural' programming language for the Amiga. Jim Butterfield, however, has had his Amiga for over two years and very little of that time has been spent programming in C. A friendly programming environment is important to Butterfield, and C, though powerful, has never been called friendly. He suggests that those wishing to start programming the Amiga consider using **AmigaBASIC**: "Though generally ignored, AmigaBASIC is very good – a very nice, very fast BASIC." Certainly not least among its virtues is the price – AmigaBASIC is supplied free with all models of the machine.

There are some adjustments that have to be made: "Losing line numbers is a bit of a shock at first, but you soon get used to it. Note also that AmigaBASIC provides only 7 digits of accuracy, not 10, like Commodore BASIC, and you must make provision for that in your code."

Jim points to other advantages of starting with AmigaBASIC: a chance to become familiar with the Intuition environment while working with a familiar language; good documentation and good example programs provided with the machine; and the ability to call Amiga library functions from within BASIC source code.

While Nick Sullivan, *Transactor* editor and author of TransBASIC, Music Assembler and many other programs for the 64, grudgingly agrees with Jim that AmigaBASIC may be a good place for the novice Amiga programmer to start, he feels that eventually the serious programmer does best to learn C. The data structures used by the Amiga's OS are documented as C structures, and most of the examples in the programming manuals are written C, as in fact was most of the OS.

There are currently no "C for the Amiga" type books for those new to C. Nick suggests *C Primer Plus* by Mitchell Waite, Stephen Prata and Donald Martin, published by The Waite Group. This text, though not without its faults, does provide a reasonably thorough and accessible introduction to C. Nick also recommends having the Kernighan and Ritchie "bible" on C handy for reference.

Also recommended by all those I spoke with are the official Commodore-Amiga reference manuals: the *Amiga Intuition Reference Manual*, the two *ROM Kernel Reference Manuals*, the *Hardware Reference Manual* (all published by Addison-Wesley), and the *AmigaDOS Manual*, published by Bantam Books. In every case it was suggested that the beginner start with the *Intuition* manual.

The two commercial C compilers come from Lattice and Manx. They complement each other in terms of strengths and weaknesses; neither offers the perfect solution. Both, however, are excellent compilers. Though Nick himself uses the Manx Aztec compiler, he acknowledges that beginners may find the Lattice compiler somewhat friendlier, with its more stringent type checking and its more verbose error messages. In addition, it is the "official" Commodore-Amiga compiler and was developed in cooperation with C-A. Lattice claims that their latest revision (4.0) has overtaken the Manx product in terms of the efficiency of the code it generates.

The Aztec C provides such useful features as precompiled header files and support for the other CPUs in the 68xxx family – neither of which is provided by the Lattice product.

Both compilers provide extensive documentation and a number of utilities. Neither will teach you how to program in C.

Once you have a compiler, you're up and running – you will require no other software to write programs. However, there are a number of utilities so useful to the Amiga programmer as to be deemed essential. At the top of the list, as far as Nick is concerned, is a command shell, such as the PD **csh** by Matt Dillon, or the **Shell** program produced by Metacomco. These provide command line history, editing and aliases to minimize the amount of retyping necessary. Next on the list of recommended support software is a recoverable RAM disk, such as the NV0: supplied with the Comspec RAM board, or the PD program **RRD**, by Perry Kivolowitz. These are similar to AmigaDOS RAM: device but retain their contents during a warm reboot – which saves you having to restock RAM after a crash during program development. Also recommended is a good text editor. Nick user *UEdit*, a fine programmable shareware editor by Rick Stiles.

Nick also recommends **Power Windows**, published by Inovatronics. This product lets you interactively set up the windows, menus, screens, gadgets andrequesters you want to use in your program. It then automatically generates the C, Modula 2 or Assembler source code for these Intuition structures. It can import images you have stored as "brushes" in **Deluxe Paint**, and convert them to Image structures for use in menus and gadgets. Doing this sort of thing by hand can be enormously time consuming, as I can tell you, having spent an entire evening drawing four simple gadgets on graph paper and converting them into hex data. "If Power Windows had been out right at the start," says Nick, "there might be a lot more Amiga software available *now*."

Other useful utilities mentioned by Sullivan include Metadigm's **Metascope** windowing debugger, Manx's **SLD** source level debugger (see below), **Gimpel Lint**, a source-code syntax checker by Gimpel Software, **Deluxe Paint II** by Electronic Arts and **Blitzfonts**, a text display speedup program by Hayes Haugen.

Chris Zamara, YATE (Yet Another Transactor Editor), and Sullivan's programming partner, has some advice for new Amiga programmers trying to make sense of the machine: "*Don't* read the RKM (ROM Kernel Manual) graphics documentation first! It think it was written before the Intuition manual and goes into a lot of detail that is confusing to someone just starting. Start with the *Intuition Manual*. As long as you know where to find the RastPort pointer for your screen or window or whatever, you're ready to start drawing."

Zamara had had some formal training in high level languages before he entered the 8-bit world but was new to C. His first attempt to read the *Intuition Manual* left him quite confused. However, after reading the Kernighan and Ritchie text and spending many hours poring over the Lattice C compiler manual, he gave it another try. "Once I got comfortable with structures and pointers, I reread the *Intuition Manual*. Then everything fell into place."

Larry Phillips is one of the Sysops of the AmigaForum on CompuServe. By day a mild-mannered mainframe technician, at night he communes with all manner of Commodore cultists. Larry does not share Transactor's C language bias, being instead a devotee of Modula-2, the language created by Niklaus Wirth as a successor to his earlier Pascal. Larry's advice to 64/128 programmers: "For those who are familiar with COMAL, I always recommend Modula-2, and now that I have **Benchmark** (the commercial Modula-2 compiler written by Leon Frenkel) it's the one I recommend specifically. Some books for Modula-2 I would recommend: *Modula-2, A Seafarer's Guide and Shipyard Manual*, by Edward Joyce; and *Modula-2 Primer*, by Stan Kelly-Bootle (a Waite Group book). These are aimed at beginners, and are tutorial in nature. One a little more advanced is *Modula-2, A Software Development Approach*, by Gary Ford and Richard Wiener. This one is very good, and really gets into program construction, especially in larger projects where more than one programmer is involved.

"Aficionados of Basic have the choice of going to AmigaBasic, with its very COMAL-like structures and hopeless user interface; or going to another language. Others available are Forth, APL, Fortran, C, Icon, ARexx, LISP, Pascal, Draco, Pilot, LOGO, and probably a few I've forgotten about. Of the above, PD or shareware versions are available for Forth, C, Icon, LISP (XLisp), Draco, Pilot, and LOGO.

"For the assembler types, all I can say is, good luck wading through the Amiga documentation and having to learn C to translate to a civilized program. There *is* a book called *Amiga Assembler Language Programming*, by Jake Commander. This is not the greatest assembler book I've seen, but will get them started and help them to understand the Amiga ways of doing things."

Assembly language on the Amiga *is* a rather different proposition, concurs Jim Butterfield. "68000 machine language is the same in nature (as 6502 ML) but it's in a much more complex environment." That statement lies at the heart of the difference between single user computers like the 64/128 and the multi-user, multitasking OS of the Amiga.

Those who wish to program on the Amiga cannot assume they have the machine to themselves. The Amiga's OS permits as many programs as can fit into memory at once to run simultaneously. Those programs must, therefore, behave as good Amiga citizens – requesting resources such as memory and I/O devices rather than simply commandeering them; sharing devices with other processes; and releasing resources when finished with them. As I have written in previous columns, it's rather like dealing with a pervasive, though efficient, bureaucracy.

The Amiga programmer is actively discouraged from delving into Amiga Kernel libraries directly, for good reason: CBM is constantly improving the OS. They will guarantee only one unchanging address between revisions: $00000004, otherwise known as AbsExecBase. This contains the address of ExecBase, the portal to the Exec library. All routines in Exec, including OpenLibrary(), which is used to open the other libraries, are addressed as offsets from ExecBase. The address of any routine, of any library, even of ExecBase itself, may change at some future date. For example, Dale Luck, architect of much of the Amiga's graphics libraries, has suggested that about 50K of Kickstart's current code will be removed from ROM in 1.3 and put onto the Workbench disk. This will make room in ROM for new graphics routines. Dale didn't say what these routines will be for, but I think it likely they will provide support for X-Windows. (For an explanation of X-Windows, see the last edition of AD.)

The need to coexist with other programs and the necessity of accessing system routines indirectly requires a fundamental change in attitude on the part of someone used to programming a single user machine. Rather than rolling up one's sleeves and digging in, one must learn to keep all interaction with the system at arm's length. It means playing by the rules, rather than inventing one's own. The reward is an enormous gain in power. After all, you have 256K of optimized code in Kickstart to tap, as opposed to the 16K of BASIC and Kernal ROM found in the C64.

If you do decide to pursue assembly language on the Amiga, Butterfield highly recommends *Kickstart's Guide to the Amiga*. *Kickstart* is an

English technical journal. The editors have gathered together much of the information published in the magazine, edited it and published it in book form – rather like an Amiga version of Transactor's *The Complete Commodore Inner Space Anthology*. Butterfield notes that copies are hard to get in North America at present, but efforts are underway to import them.

You will also need an assembler, of course. Charles Gibbs's **A68k** *is a PD assembler that is compatible with the assembler include files found in the programmer's support disks, which are available from CBM West Chester. Alternatively, these files are also included with commercial assemblers from Metacomco, HiSoft, Lattice and Manx, to name a few.*

AmiExpo

I shall finish up this edition of AD with the promised report on the AmiExpo show held in New York, October 10–12. I was unable to attend but Nick and Karl went and returned bearing a stack of press releases. Attendance was over 8000 for the three days, with more than 50 vendors exhibiting. A brief summary of products of particular interest to readers of the Transactor (or at least, of interest to *me*) follows. All prices are in US dollars.

Byte by Byte were offering a self-powered RAM expansion board for the A500: $299.95 unpopulated, $699.95 fully populated with 2 Mb of RAM. Memory checking software is included. . . Discovery Software International announced a C-Shell for the Amiga called **Amnix** that provides over 40 resident commands, command line history and editing, and environment variables for $49.95. They also announced **DX-16** and **DX-11**, Amiga emulations of the Hewlett-Packard HP-16C programmer's calculator and the HP-11C scientific calculator respectively. Both fully multitasking programs are available for $49.95. . .

Synthia is a digital synthesizer program for creating IFF instruments. It uses a variety of techniques, including additive, interpolative and subtractive synthesis, and can modify existing instruments with such sonic brushes as true reverb, comb filtration and waveshaping. It includes an IFF music player and appears to be a very powerful package. It's yours for $99 from The Other Guys. . . Other music software demoed at AmiExpo included **The Sound Quest**, a Roland D-50 synthesizer editor/librarian package by Sound Quest Inc. of Toronto; and Roger Powell's long-awaited **Texture** port to the Amiga.

ASDG Inc. had a very strong presence at the show. Of particular interest was the **Satellite Disk Processor**, an interface for the A1000 and A2000 that will support up to 56 SCSI devices and two ST-506 devices. ASDG claims the card uses its onboard 68000 CPU, 512 Kb cache, DMA and MMU to provide over "400 Kb per second average user through-put". Also announced were further details on the **2000-and-1** expansion box for the A1000. This unit provides two Zorro I slots (for cards designed to the original Zorro spec), five Zorro II slots (for all those great A2000 cards), three IBM PC/AT slots (for the BridgeCard and other PC/AT cards), one coprocessor slot (for the 68020/68881), two 3.5 inch drive bays, one 5.25 inch drive bay and a 200 watt power supply. The only thing it doesn't provide, at $799, is the A2000's video slot. . .

Spirit Technology showed their **Inboard** internal memory expander for the A500. The unit plugs into the 68000 socket and can provide up to 1.5 Mb of RAM and a battery-backed clock/calendar. The unpopulated board sells for $279. There have been reports of severe damage to the machines of some of those who have installed this board so *caveat emptor*. It should be noted that Spirit Technology "guarantees Inboard to be the finest and most reliable design available or your money back". Whether this is a guarantee that the unit will actually *work* is open to interpretation. . .

Manx's Jim Goodnow was showing the new Aztec C **Source Level Debugger**. This product will answer the prayers of many, me not least. It provides a facility called "back tracing", which allows the user to display all active function names and the values of passed parameters. In addition, active frame context switching makes it possible to examine the variables that are visible from any active function, and reusable command macros can be used to customize the debugging environment. The windowing capabilities of SDB let the user display C source and command output separately, with a third window for entering commands. All of these goodies and more are available for $75. Manx also announced version 3.6 of their C compiler and declared that a future release would include an ANSI standard compiler.

Spencer Organization, Inc. showed **APL.68000** for the Amiga. According to the company, the **APL Interpreter** supports APL multitasking (whatever that is), is fully 'Intuitionalized' (my expression), is provided with a full interface to Amiga graphics functions and has a built in APL/ASCII terminal emulator. All this and more, folks, for only $99. . .

Meridian Software Inc., publishers of **Zing!**, were selling **The Demonstrator**. If you remember my description of the PD program **Journal** a couple of issues ago, this program will ring a bell. It will record one's interaction with the Amiga and play it back upon demand. One can apparently add text windows and speech to create elaborate demos or tutorials. In addition, one can control the speed of the playback. The price: $39.95. . . **Galileo** sounds much like an Amiga version of the excellent **Sky Traveller** program for the C64. It can be used to view the sky from any point on earth for any date this century and displays the constellations in nine different levels of brightness. The program was developed by Mike Smithwick, an astronomer and computer graphics artist currently working for NASA's Ames Research Center. You can see the stars on your Amiga for $69.95. . .

That's all, folks!

I am sorry to say that this will be the last edition of Amiga Dispatches written by me. Due to a number of recent changes in my personal life, I no longer have the time for the many hours of research this column requires. Rather, it will be spent on more important matters – my wife Cate and my young son Alex.

I *will* be contributing the occasional article to the *Transactor for the Amiga*: first on the list is the Amiga 2000 review I have promised for so long. It will include a look at the B2000, otherwise known as the West Chester 2000. I have yet to finish putting it through its paces, but I can tell you this: it's a significant improvement on the West German 2000.

I will be placing the AD keyboard into the capable hands of Don Curtis from Denver, CO. His first column will appear in the premiere issue of TA. In the meantime, let me know what you think: of life, love or the 68030. I have drawn a great deal of pleasure from the electronic and paper letters I have received while writing AD. I hope you will continue to include me in your Amiga adventures. I can be reached at *The Transactor* or electronically via:

CIS: 71426,1646
PeopleLink: AMTAG
BIX: dispatcher
GEnie: t.grantham
Bloom Beacon BBS: Tim Grantham
(416 297-5607)

Change The Mouse Pointer in AmigaBasic

Anthony Bryant
Winnipeg, Manitoba

. . .a short program to change the default mouse pointer to one of your own design. . .

Basic programmers may wish to use a custom pointer in a program, or C or assembler programmers may wish to try out a new pointer design (or sprite design) quickly and interactively. With AmigaBasic's LIBRARY statement, you can use system functions besides those available directly with Basic commands. Intuition's SetPointer() and ClearPointer() functions will be used in this program to switch to a new pointer and back to the default again.

Using Libraries in AmigaBasic

The arrow pointer is the default pointer used by Intuition so to change it ("it" in hardware is sprite 0), we have to "get into" Intuition. We can call Intuition functions, as we can call functions in other system libraries, by using the LIBRARY command. The LIBRARY command, given the name of a library (in this case "intuition.library"), will make the functions of that library available from AmigaBasic. The catch, however, is that for LIBRARY to work, it needs the ".bmap" file associated with the library being used, which gives Basic information about the functions in the library, like their addresses in memory and what parameters they accept.

Since the information contained in a ".bmap" file is specific to a particular version of the operating system, the file should be custom-made for the system you're using. The program "ConvertFD" in the "BasicDemos" drawer on the AmigaBasic disk (version 1.2) will make a ".bmap" file from a ".fd" file, which contains more general information about functions within a library. The ".fd" files for all libraries can be found in the "fd1.2" directory on the AmigaBasic disk. You can create the "intuition.bmap" file that you'll need with the command:

```
ConvertFD :fd1.2/intuition_lib.fd
```

You'll also need the .bmap file for the exec library, so run ConvertFD on the file "exec_lib.fd" in the fd1.2 directory as well. You should keep these files around for future programs; a good idea is to make a "libs" directory on your AmigaBasic disk and put all your .bmap files there. You can then use them with the LIBRARY statement by specifying the full pathname of the files, like:

```
LIBRARY ":libs/intuition.library"
LIBRARY ":libs/exec.library"
etc.
```

(Assuming that your current directory when you run the Basic program is somewhere on the same disk.)

Once the LIBRARY command has been used to link you with the system library of your choice, you can call the routines by name, passing them parameters just as in a C program. Routines that return values must be declared with:

```
DECLARE FUNCTION FunctionName LIBRARY
```

using a '&' at the end of the function name if it returns a long word (as most system functions do).

You've probably scanned the several demo programs that use the LIBRARY statement to see how it works, and you may have looked through the fd1.2 files to familiarize yourself with the names of all those system functions that you are so eager to try. But you really need the manuals – Intuition and ROM Kernal – to make sense of the libraries.

Intuition: The Manual

Chapter 4 of the Intuition manual explains the functions used to create a custom pointer in a window:

```
SetPointer (Window, Pointer, height, width, xoffset, yoffset)
```

Changes the sprite definition in the given window, and

```
ClearPointer (Window)
```

Restores the default pointer in the given window.

The program that follows is a simple translation to AmigaBasic. It's basically a matter of passing SetPointer() the right parameters, then calling ClearPointer() to set the pointer back to normal again. We get the pointer to the Window with the WINDOW(7) function, and the left, width, and offsets for the pointer are just part of the sprite definition in the DATA statements.

Passing SetPointer() the pointer to the sprite definition data itself is bit trickier. Perhaps the easiest method would be to store the sprite definition in an integer array and pass SetPointer a pointer to the array via Basic's VARPTR function. The trouble with this approach is that sprite definitions, like any data that needs to be accessed by the graphics or sound hardware, must be in "chip" RAM. An unexpanded Amiga has chip ram only, so using an array would work fine, but the program would stop working if extra RAM was added to the system. To do the job properly, memory is allocated with the AllocMem() function in the Exec library – we can tell AllocMem() to give us chip RAM. The sprite definition is stored in the chip ram with a POKEW (POKE Word) statement in a FOR. . .NEXT loop.

Before the program ends, the memory that was allocated is released back to the system with FreeMem(). If this is not done, the memory allocated will be lost until the next system re-boot.

To use the AllocMem and FreeMem functions, a LIBRARY command for "exec.library" must be used. Also, since AllocMem returns a longword result, it must be explicitly declared with DECLARE FUNCTION.

When you run the program (Basic's main "run" window should be active when you do this), it will display the new pointer until you click

the left mouse button, at which point it will switch back to the default arrow (or whatever you've set in Preferences) and the program will end. Not much use in itself, but you can use the code to attach custom pointers to windows in your own programs.

Change the RESTORE command to try any of the four pointers defined in the DATA statements, or try changing the data to make pointers of your own. To make your own pointers, you'll need to know a bit about how sprite definitions work.

Creating Sprite Definition Data

The data that constitute the sprite definition are made up of two words per screen line for position and colour information. (The first and last two words in the definition do not actually describe the sprite, and should be zero in all definitions – the system will use these words for its own needs.)

Both words together describe the colour of each pixel on a single horizontal line of the sprite. A word is 16 bits, the maximum width of a sprite. The first word contains the LSB (least significant bit) data and the second word has the MSB (most significant bit) data; the two bits specify one of three possible colours, or transparency:

	msb	lsb	
Colour	0	0	is transparent (blue)
Colour	0	1	is med intensity (red – hardware colour reg. 17)
Colour	1	0	is low intensity (black – hardware colour reg. 18)
Colour	1	1	is high intensity (red – hardware colour reg. 19)

A few examples of custom pointers are given in the program to show how the colours work (the data is in hexadecimal). XPointer is a direct copy from the example given in C in the Intuition manual; XPlain is a one-colour version.

```
'** Change mouse Shape via Intuition Custom Pointer **

DECLARE FUNCTION AllocMem& LIBRARY
LIBRARY "intuition.library"
LIBRARY "exec.library"

chip% = 2  'to tell AllocMem() we want chip ram

RESTORE XPointer  'which pointer we want

READ pHeight%, pWidth%, pXOffset%, pYOffset%
pSize& = 2 * (2 * pHeight% + 3) '# of bytes required for sprite definition
mem& = AllocMem&(pSize&, chip%) 'allocate some chip ram

'** copy sprite data into chip ram **
FOR i& = mem& TO (mem& + pSize&) STEP 2
 READ x%
 POKEW i&,x%
NEXT i&

'** switch to our new pointer **
CALL SetPointer(WINDOW(7),mem&,pHeight%,pWidth%,pXOffset%,pYOffset%)
PRINT "SetPointer"
PRINT "(Click left mouse button to ClearPointer)"

'** wait until user clicks left mouse button **
WHILE MOUSE(0) <> -1
WEND

'** switch back to normal pointer **
CALL ClearPointer(WINDOW(7))
PRINT "ClearPointer"

'** free memory and close libraries **
CALL FreeMem(mem&, pSize&)
LIBRARY CLOSE
END

' sample custom pointer data follows

XPointer:  'three-colour pointer
DATA  9, 9, -5, -4
DATA  &H0000,  &H0000
DATA  &HC180, &H4100
DATA  &H6380,  &HA280
DATA  &H3700,  &H5500
DATA  &H1600,  &H2200
DATA  &H0000,  &H0000
DATA  &H1600,  &H2200
DATA  &H2300,  &H5500
DATA  &H4180,  &HA280
DATA  &H8080,  &H4100
DATA  &H0000,  &H0000

XPlain:  'one-colour (colour 01 only)
DATA  9, 9, -5, -4
DATA  &H0000,  &H0000
DATA  &H8080,  &H0000
DATA  &H4100,  &H0000
DATA  &H2200,  &H0000
DATA  &H1400,  &H0000
DATA  &H0000,  &H0000
DATA  &H1400,  &H0000
DATA  &H2200,  &H0000
DATA  &H4100,  &H0000
DATA  &H8080,  &H0000
DATA  &H0000,  &H0000

SPointer:  'three-colour separated
DATA 9, 9, -5, -4
DATA  &H0000,  &H0000
DATA  &H0FC3,  &H0000
DATA  &H3FF3,  &H0000
DATA  &H30C3,  &H0000
DATA  &H0000,  &H3C03
DATA  &H0000,  &H3FC3
DATA  &H0000,  &H03C3
DATA  &HC033,  &HC033
DATA  &HFFC0,  &HFFC0
DATA  &H3F03,  &H3F03
DATA  &H0000,  &H0000

BPointer:  'two-colour box (colour 01 and 11)
DATA 13, 16, -8, -6
DATA  &H0000,  &H0000
DATA  &HFFFE,  &HFFFE
DATA  &HC106,  &HC006
DATA  &HC106,  &HC006
DATA  &HC106,  &HC006
DATA  &HC106,  &HC006
DATA  &HC106,  &HC006
DATA  &HFFFE,  &HC006
DATA  &HC106,  &HC006
DATA  &HC106,  &HC006
DATA  &HC106,  &HC006
DATA  &HC106,  &HC006
DATA  &HC106,  &HC006
DATA  &HFFFE,  &HFFFE
DATA  &H0000,  &H0000
```

FACTS BEHIND THE FLASHES
The Amiga Startup Messages

By Betty Clay, Arlington, Texas

It is not unusual for a computer to try to communicate with its owner during the start-up routines. Remember the flashing of the lights on the 4040 and 8050 disk drives? On my 8050 drive, two flashes of the green LEDs indicate that things are fine. More than two indicate a problem, and if there were such a thing as a local Commodore repairman, he would be helped if he knew how many times my drive was flashing the lights.

Another way the earlier Commodores communicated with us was by printing the number of bytes free on the screen at startup. If the number was wrong, we knew that some of the RAM had failed to accept data during the startup routine, since the computer would assume that the end of BASIC was at the last location in which it could write and read back data. If a memory chip became bad, the number of bytes free would be the same as the number to which the processor had successfully written and read back its startup data.

The Amiga has a rather elaborate set of diagnostics, if we only know how to interpret them.

THE AMIGA START-UP ROUTINE

When you turn on your Amiga, it has a rather long and complicated set of routines through which it must go before it can allow you to interrupt it. As the startup process goes along, Amiga is trying to let you know whether all is well. There has been a small problem with this, however – Commodore forgot to tell us what the signals mean!

Here is a list of the start-up routine activities:

1. Clear all of the chips of old data.
2. Disable DMA and interrupts during the test.
3. Clear the screen.
4. Check the Hardware (make sure the 68000 is working)
5. Change the screen color to show whether this test was passed
6. Do a checksum test on all the ROMS
7. Change screen color to show if ROMS passed the test
8. Begin the system startup
9. Check the RAM at $C0000, and move SYSBASE there
10. Test all of the chip RAM
11. Change the screen color to show if the RAM passed the test
12. Check to see if software is coming in OK
13. Change the screen color to show if the software test is passed.
14. Set up the chip ram to receive data.
15. Link the libraries.
16. Check for additional memory and link it in if found
17. Turn the DMA and Interrupts back on.
18. Start a default task.
19. Check to see if the computer is using a 68010, 68020, and/or 68881.
20. Check to see if there is an Exception (processor error)
21. If so, do a system reset.

AND THE MESSAGES ARE IN TECHNICOLOR!

As this routine is taking place, the Amiga is sending you messages with the screen colors. If all is well, we usually see this sequence:

Dark gray	The initial hardware tested OK. The 68000 is running and the registers are readable
Light gray	The software is coming in and seems OK
White	The initialization tests were all passed

But if something is wrong with your Amiga, you might see:

Red	If there is an error in ROM
Green	If there is an error in the Chip RAM
Blue	If an error was found in the custom chips
Yellow	If the 68000 found an error before the error trapping software (the guru) was up and running

The most likely of these errors seems to be the error in Chip RAM. Only this week, I saw an AMIGA 500 flash a brilliant green screen when an expansion RAM board was put in hastily, and did not settle in correctly. A repositioning of the board corrected the problem in that case. I have not yet seen a red, blue, or yellow screen indicator.

KEYBOARD MESSAGES

The Amiga keyboard is not as dull an object as I had originally thought, either. It contains a processor of its own – a Rockwell/NCR/MOS Technology 6500/1. It also has 2K of ROM, 64 bytes of RAM, and four I/O ports of eight bits each. There is a built-in crystal oscillator running at 3Mhz, also. All but the very earliest of keyboards also have a "watchdog timer" which will reset the keyboard's processor if it stops scanning the keyboard for more than 50 milliseconds.

It is possible for the computer to be powered up before the keyboard is plugged in, in which case the keyboard will have to go through its self-test after it is connected to the computer. Most of us, however, will have the keyboard attached, and the self-test will take place while we are watching the screen, changing disks, etc.

The keyboard self-test consists of four steps. First it does a checksum on all of the ROMs. Then it checks the 64 bytes of RAM, and then the timer is tested. Then the keyboard must achieve proper synchronization with the computer. It does this by slowly clocking out 1 bits until it receives a handshake pulse from the computer. Once this pulse is received, the keyboard must inform the computer of the results of its self-test. Should the self-test fail, the code for failure can be sent to the computer without waiting for the handshake pulse.

IN CASE OF FAILURE

After informing the computer that the self-test has failed, the keyboard will then try to notify the user that it is in trouble. This is done by blinking the CAPS-LOCK light. Here is the code:

One blink	The keyboard ROM check failed
Two blinks	The keyboard RAM check failed
Three blinks	The watchdog timer test failed
Four blinks	A short exists between two row lines or one of the seven special control keys

The last check had not been implemented at the time my ROM Kernal Manual was printed, but was in the plans. It would be unusual for the user to have typed anything during this self-check time, but if any keys have been depressed, the codes for those keys would then be sent to the computer, a "terminate key stream" code would be sent, and then the CAPS LOCK LED shut off, indicating the end of the keyboard startup sequence.

Should you be so unfortunate as to have your Amiga get into difficulties, perhaps these codes will help you and your repair man to put it in good health again.

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

New Editorial Assistant Mends Our Ways

There's just been an important addition to the staff at The Transactor, and that means good news for authors and anyone else sending mail to our editorial department. Moya Drummond – our new editorial assistant – is on the job now, and she's correcting some of our bad habits, mostly in the area of responding to mail. If you sent us an article recently, chances are you've either heard from Moya or will be hearing from her soon, just to let you know we received it. You will also find out the fate of the article once we decide, which means you'll know if we plan to use it *before* you see it in print – correcting another nasty habit that we're happy to lose. Moya is doing wonders for organization around here, and if you have questions about anything you've sent to the editorial department, chances are she'll be able to help you out. Questions about advertising in the Transactor should also be directed to Moya (like everyone at the T., Moya wears many hats). We thank you for bearing with us in the past, and if you've tried to communicate with us on something but given up, please give us another chance!

Avoid Duty and Federal Taxes on Quick Brown Boxes

We've recently made arrangements to supply Quick Brown Box battery backed RAM cartridges via Transactor mail order. In Canada the prices are identical to the US price plus the exchange – no duty or 12% FST! Ontario residents must still add 7% provincial sales tax though.

The Quick Brown Box is a proven product that we're rather pleased to make available. For more details see the Mail Order section of News BRK. For more information than that, call Brown Boxes Inc. at (617) 275-0090.

Combination Magazine Subscriptions

In case you haven't heard, the premiere issue of *Transactor for the Amiga* will be released this January. Subscription prices are identical to the Transactor Classic (as it's become known around the office). Depending on when this issue hits the streets, you may still have time to subscribe at HALF the regular price. Offer ends January 1st, 1988. (Sorry, but Herman insists).

Disk subscriptions for the Amiga magazine will be regular price after January 1st, too. We'd like to point out, however, that there are still a couple of ways to save. With combination subscriptions to Transactor Classic and the 5¼" disks we offer a choice of free merchandise, but no price break. With combination subscriptions to T/A and the T/A Disk, we offer a price reduction on the total, but no merchandise selection. You can also get a combination break on a subscription to both magazines. We're not offering "cross-combination" price breaks – that would make things too complicated (e.g. T/A with T Classic Disks, or T Classic with T/A Disks). But, if you order subscriptions to both mags AND both disks, you not only get a break on the total, but you're also entitled to a FREE T-Shirt or any other item on the list at the top. Check out the order card at center.

If you're already subscribing to Transactor Classic, we're allowing the addition of T/A subscription items for *the difference* in price. So if you're in the U.S. and you're getting T Classic, the combo price would be $27.00, minus $15.00, equals $12.00 for adding a sub to Transactor for the Amiga. Same applies to all combos for "Both T & T/A" (see card).

Subscription Switch

With the *Transactor for the Amiga* coming this January, many of our readers will want to switch to the new mag for the higher concentration of Amiga material. To switch to the new magazine, there's no charge. Simply put your name and subscription number on our postage paid card and check off the appropriate box. Please don't omit your name as this gives us a cross-reference to ensure we change the correct subscriber record.

Originally we planned to impose a February deadline for switching subscriptions. Then we considered the situation where a reader might not get an Amiga until March and disallowing the transfer would be unfair. Officially, therefore, the subscription switching option will never be discontinued.

Problems Keep Life Interesting

There's a first time for everything and last issue was our first to be shipped in poly bags. One reason for the bags was to protect the magazine against the hazards of door-to-door delivery. Another was to keep the mailing label *off* the cover. The first reason panned out rather well, but the second didn't. Ooops. Please accept our apologies – the labels *will* be glued to the outside of the bag from now on.

When we looked into the poly bag idea, we just assumed we would be printing the labels on paper, as we had always done. Then we made a mid-stream modification and decided to print the labels on sticky-backs to make it easier to complete our business reply card for things like orders and renewals. Please don't throw this away when you get your copy. Even if you have no intention of sending us the order form in this issue, the reply card makes a tidy spot for the label. It means your magazine will carry *your* ID in case you lend it out, and if you should move, your change of address notice is already half completed – just add your new address and pop it in the mail. It's also a good place to keep a record of your subscription number and expiry date, all of which will be lost if it makes it to the trash bin.

Early Renewal Notices

We also had a report that a subscribers' first copy arrived containing a renewal notice. So far there's only been one case reported so it could have been a fluke. However, if it ever happens again, even though it won't (fingers crossed), always check the top right corner of the mailing label for the official expiry issue. The data printed there is extracted directly from two fields in our data base. The same two fields are used by our "expiry algorithm" to determine when a subscription gets discontinued. A separate program determines who should be sent a renewal notice. But in this case, neither was responsible – in fact, our poly bagging service has asked that we place the blame on him, but we don't like doing that.

Two Separate Subs – Mag Not Included With Disk.

Our last issue detailed two incredible offers regarding the new Transactor for the Amiga. After the sentence that listed the prices for a disk subscription there was a sentence that read, "You'll get a disk with every magazine containing all the programs. . .". This has been misinterpreted by some as meaning the price of a disk subscription *includes* a magazine subscription. We thought we made it clear that magazine and disk subscriptions carried separate prices, but apparently not. We've been selling disk and mag subs separately for so long that we didn't anticipate the potential for assuming one would be included with the other. Our apologies. Perhaps that sentence should have read, "Magazine subscribers who also become disk subscribers will receive a disk of the Amiga programs published in the corresponding magazine, and we'll probably add extra programs too.".

Half Price For One Year Only

Several of the subscription cards we received during our half price offer on T/A arrived with two and three year orders. However, the offer was never meant to cover multiple year subscriptions. Some of our colleagues insist we've gone mad making the offer for even one year! We *will* apply all your cash to subscription magazines, but those who ordered two years at half price will receive 1½, and those who ordered three years will get two. And, we also apologize for overlooking this possible assumption.

Office Access

Since the announcement of Transactor for the Amiga, our phones are chiming off the pressure sensors! To keep our production and service operations functional, we've decided to limit calls to Mondays, Wednesdays and Fridays. So if you have an order or subscription problem, please try to have it on one of these days. Once we have some wrinkles ironed out and things return to a nice, tame chaos around the office, we hope to lift our office access limitation.

The 20/20 Deal

. . .is still in effect – order 20 subscriptions to the mag or disk (either T Classic OR T/A), 20 back issues, 20 disks, etc., and get a 20% discount. (Offer applies to regular prices and cannot be combined with other specials)

No Longer Available

Our 1541 Upgrade ROM Kit, which eliminates the SAVE@ bug plus a few others, is now discontinued. If you would still like to obtain a set, complete instructions are in an

article published in Volume 7 Issue 02 and Disk 13 contains the ROM image file you'll need to burn your own EPROMS. However, we're reasonably sure that the ROM image on disk is compatible with the 1541 *only*. 1541**C** owners will need to create an image of their ROM set, then make the changes described in V7 I02, but with minor adjustments to accommodate for what are more than likely simply slight address changes. Please let us know if you do one so we can print an article update!

TPUG No Longer Supplying Transactor

As most of you know, for the past seven issues we've been supplying Transactor with TPUG inserts to TPUG who have been sending them to their members as their regular club periodical. Since starting, TPUG's order has fallen substantially and we eventually told them we would have to raise our prices. Shortly afterwards, TPUG asked that we supply a mixed order of Transactor and Transactor for the Amiga, which would have meant that the order for each publication would drop again. TPUG wasn't prepared to cover the additional costs this would entail, and instead decided to decline further orders altogether.

As we go to press we do not know what substitute TPUG will offer. However, this will in no way affect your TPUG membership. Members will still have access to TPUG's vast software library and regular members will still have a full schedule of meetings to attend.

New and Improved Transactor Disks!

Transactor Disks 1 through 19 have been totally re-mastered. Every bloop from every issue has been corrected for every disk. The directories have been tidied up a little too, with "directory placemarkers" so you can see where the "BITS" listings end and where the programs related to articles begin. Otherwise, the disks still contain the same programs as before.

The TransBASIC series appeared in 12 issues, starting with Vol. 5 Issue 05 and ending with Vol. 7 Issue 04. Each Transactor Disk always carried forward the TransBASIC modules published in all previous issues, and the new modules were added. Disks 4 to 15 now contain "TransBASIC Samplers". Each disk in this range now carries only the modules published in the corresponding issue plus everything you need to try them out. Originally you needed an assembler like PAL or SYMASS to install the commands published in TransBASIC columns. SYMASS 3.13 has been included on Disks 4 through 15 and is automatically loaded by the file "transbasic.run", also on all of these disks. This boot program displays enough instructions to get you started, and proceeds to set up the sampler system. Give it a try!

Probably the most important reason for re-mastering the disks was to make the new custom labels. All Transactor Disks now have typeset three-colour labels. The colours are co-ordinated to the same colours of the corresponding magazine, and besides the Disk number, we've included the publication date, volume and issue number, and the issue theme where applicable. Best of all, the entire directory of each disk is listed right on the label! It sure beats loading and/or listing directories trying to find that one elusive program! (Actually, I think *I'm* going to enjoy having them more than anybody).

We regret to say that we have not devised any plan for upgrading Transactor disks because the fee couldn't be much less than the price of brand new ones. However, if there's enough demand we may sell sets of the colour labels. The labels wouldn't match the disk directories *exactly*, but all file names were left unaltered and the new labels would be quite useful. Let us know what you think!

Fish Disks With Custom Labels

Starting in January, with the premiere issue of Transactor for the Amiga, we'll be supplying the complete library of Fish Disks. Fred Fish has assembled an impressive collection of public domain software and shareware for the Amiga and they're available from several sources. However, on the Fish disks you'll receive from us will be a custom label. Just like Transactor Disks, a condensed version of the disk directory will be listed right on the label! When space permits, we'll also print short descriptions next to each program name. No more reading endless directories off disk when you're searching for that one file you need! To make organizing your library even easier, we plan to use a number of different colours (i.e. readable colours). An order form and prices will appear in every issue of Transactor for the Amiga.

Transactor Bi-Monthly Special Extended

Because this magazine is coming out so soon after our previous one, we've decided to extend our bi-monthly special from last issue. To recap, order any back issue at the regular price of $4.50 (US/C), and get additional back issues for only $2.00 each! Order 10 total and the effective price per copy is cut by half! ($4.50 + 9 × $2.00 = $22.50). We actually had another special all picked out before we chose to extend this one. The extension means it will apply only to orders postmarked before March 1, 1988.

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.

■ Quick Brown Box - Battery Backed RAM for C64 or C128
■ 16K QBB - $ 69.00 US, $ 89.00 Cdn (Please add $3.00 US,
■ 32K QBB - $ 99.00 US, $129.00 Cdn $4.00 Cdn P&H
■ 64K QBB - $129.00 US, $169.00 Cdn for these items)
■ QBB Utilities Disk - $6 US, $8 Cdn (post paid if ordered with cartridge)

The Quick Brown Box cartridges for the C64 and 128 can be used to store any type of programs or data that remains intact even when the cartridge is unplugged. Unlike EPROM cartridges, the QBB requires no programming or erasing equipment except your computer. Loader programs are supplied and you can store as many programs into the cartridge as its memory will allow. It may even be used as a non-volatile RAM disk. Auto-start programs are supported such as BBS programs and software monitoring systems that need to re-boot themselves in the event of a power failure. All models come with a RESET push button and use low current CMOS RAM powered by a 160 MA-Hr. Lithium cell with an estimated life of 7 to 10 years. Comes with manual, and software supplied includes loader utilities and Supermon+64 (by permission of Jim Butterfield). 30 day money back guarantee and a 1 year repair/replacement warranty.

■ 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.

TransBASIC II contains 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 *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 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)
■ Vol. 8, Issue 04 – Feature: Transactor For The Amiga (■ Disk 21)
■ Vol. 8, Issue 05 – Feature: Binary Trees (■ Disk 22)

Industry News

Parents Legally Responsible for Teenage Pirates?

New York – Can parents be held legally responsible for acts of software piracy by their teenage children? Jonathan D. Wallace, a computer lawyer representing the plaintiff in Weaver v. Doe, a case pending in federal court in New York, believes they can.

Weaver, the plaintiff, owns the copyright of "Cards", a commercially distributed card-playing simulation for the Atari ST. The teenage defendant allegedly operated a pirate bulletin board system from which users could download "Cards" and other

copyrighted programs. Although software companies have sued software pirates before, this is the first case of which Wallace is aware in which the pirate's parents have also been sued. Wallace believes the case raises a question of first impression under the copyright law. "Our argument is that a parent who supplies the computer equipment and telephone line which is used to operate a pirate bulletin board, and who then tolerates the trading of pirated software, contributes to the copyright infringement", Wallace said. "Since teenagers usually have no assets with which to pay a judgment, holding the parents responsible will give a strong incentive to families not to condone this type of behaviour."

For further information contact: Jonathan D. Wallace, Meatto, Russo, Burke & Wallace, 747 3rd Avenue, New York, NY. 10017, telephone (212) 759-0523

The 64 Emulator for Amiga

Last issue the phone number for ReadySoft was incorrect as published. For more information about the 64 Emulator contact: ReadySoft Inc., P.O. Box 1222, Lewiston, NY, 14092, (416) 731-4175. Please note, the phone number is for ReadySoft's headquarters in Richmond Hill, Ontario. Our apologies for any inconvenience.

THE ACCOUNTANT v2.0 for C128

THE ACCOUNTANT is a 4-part, menu-driven accounting program. Accounts Payable and Receivable, Payroll and General Ledger are on a single disk. Although no longer able to run on the 1541 disk drive, it has versions ready to run on the 1571 and the new 1581 disk drives.

New features include a Disk File that will "rebuild" most corrupted files; a rewritten Payroll section for 100 employees; individualized State Withholding Tax Rates; and a third percentage reduction. Complete details of all deductions are maintained on a monthly, quarterly and annual basis. The program produces over 20 CPA style reports on every aspect of a business.

THE ACCOUNTANT will be the forerunner of a new accounting system from KFS for the Amiga 500 and 2000, scheduled for release January 1st.

KFS Software Inc., P.O. BOX 107, 1301 Seminole Blvd. Suite 117, Largo. FL 34649-0107, USA.

New Utility Program For the 1581 Disk Drive

Free Spirit Software Inc. has released Super 81 Utilities – a full-featured utilities system for the Commodore 1581 Disk Drive and Commodore 128 computer. An 80 column monitor is required.

Features include:

- Copy whole disks from 1541 or 1571 format to 1581 partitions
- Copy 1541 or 1571 files to 1581 disks
- Copy 1581 files to 1571 disks
- Backup 1581 disks or files with one or two 1581s
- 1581 disk editor with simultaneous display in hex or ASCII
- 1581 drive memory monitor and RAM writer
- Perform many CP/M utility functions
- Perform numerous DOS functions: rename a disk, rename a file, scratch or unscratch files, lock or unlock files, create auto-boot etc.

The program is supplied on both 5¼" and 3½" diskettes for either 1571 or 1581 drives, which can be utilized as device numbers 8 or 9.

Super 81 Utilities is available from Free Spirit for $39.95. Shipping & Handling are free. The program disks are not copy-protected. More information from Joe Hubbard, Free Spirit Software Inc., 538 S. Edgewood, La Grange, IL 60525, USA, telephone (312) 352-7323.

Survey-Master for C64 & C128

A market survey facility for the C-64/C-128 user. This program establishes survey parameters and analyses the results. It allows for different sample sizes and determines what effect they will have on the confidence one may place in the final data (the confidence interval).

SURVEY-MASTER uses the sample size and survey data to generate screen and printed reports. If the data consists of numbers, the report includes the average, standard deviation and standard error of the mean and confidence interval. If the data consists of yes/no or option/brand preferences, the reports include standard error of the percent and confidence interval.

The program automatically corrects for large and small samples; its reports recap all the analysis criteria (confidence level, sample size, population size); built-in T-Tables are featured allowing correction to results obtained from relatively small samples at confidence levels of 70, 80, 90, 95, or 98 percent. It is compatible with the C-64/C-128 single or dual 1541 disk drives and 1525-emulating printer.

Available at $29.95 from Strategic Marketing Resources Inc., P.O. BOX 2183, Ellisville, MO 63011, telephone (314) 256-7814.

Satcomm 64

A new satellite tracking program for the many licensed amateur radio operators who use the C-64 or C-128 for communicating in RTTY, ASCII and CW modes. Features: a master menu allows quick activity selection (12 options); information on up to 15 different satellites can be stored; quickly confirms W1AW Reference Orbits; a single entry of the time bracket during which the user is available will allow a printed report of up to 31 days of access times (during the specified time bracket) for any satellite; the same one-time entry can also produce a report for any given day of the access times for up to three different satellites of interest.

Added features include an easily changed satellite menu (together with associated frequency and Keplerian element data); choice of screen plus printed report, or screen alone; easily altered user defaults (start time, time increment, etc.); and ability to override defaults by simply making an entry.

While the Commodore is performing real-time tasks SATCOMM-64's pre-printed reports include satellite azimuth and elevation, altitude, longitude and latitude, local time, UTC day, geographic areas that are within the satellite's communication range, Doppler shift, minimum and maximum communication distance, operating frequencies, orbit number and phase.

For C-64/C-128 users who are not amateur radio operators, the program may be used to track the current group of easily visible satellites including Salyut-7, MIR and Cosmos 1870, NOAA and Meteor. The program is compatible with the 1541 disk drive and any 1525 emulating printer.

Available from Strategic Marketing Resources Inc., P.O. BOX 2183, Ellisville, MO 63011, telephone (314) 256-7814.

Precisely and Quarterback for the Amiga

From Central Coast Software a new wordprocessor called PRECISELY and a fast hard disk to floppy back-up utility called QUARTERBACK for Commodore Amiga users.

PRECISELY delivers fast printer speed and screen updates and features a non-technical user interface, ease of use and inexpensive price. It accepts documents in the formats of PaperClip, SpeedScript and Pocket Writer. It will conveniently print selected screen areas such as an address for use with an envelope; supports multitasking, multiple windows, keyboard macros, online help, oops key to undo mistakes, column cut and paste, and many other features.

It sells for $79.95 plus $3.00 shipping charge.

QUARTERBACK transfers 20MB to floppy in forty-five minutes; supports full/subdirectory/incremental backup and restore, with automatic formatting of diskettes, automatic catalog of files, and automatic diskette sequence numbering and checking. It provides graceful error recovery, runs with Workbench or CLI, is multitasking, works with all AmigaDOS compatible hard disk drives, and isn't copy protected.

Available for $69.95 plus $3.00 for shipping.

For more information on PRECISELY or QUARTERBACK contact Central Coast Software, 268 Bowie Drive, Los Osos, CA 93402, telephone (805) 528-4906.

Legal Care For Your Software

Nolo Press has brought out a completely revised and updated 3rd edition of Legal Care for Your Software. In the last five years a number of important legal developments has occurred including:

- the "look and feel" of the screen format of a software program is now protected by the copyright law
- the overall structure and logic of program code is now considered protected by copyright
- freelance programmers may now be considered employees for purposes of the "work for hire rule"

• microcode (the instruction sets on microprocessor chips) has been held to be protected under the copyright law

Authors Daniel Remer and Stephen Elias incorporate these and hundreds of other smaller but significant changes; address the legal concerns of both software developers and publishers; explain copyright, patent and trade secret laws and what to do if an infringement occurs.

Available from Nolo Press, 950 Parker Street, Berkeley, CA 94710, telephone (415) 549-1976.

MIDI Interface for C64 & C128

MIDI 64 is an all-Canadian intelligent MIDI interface with innovations such as a 16K auto-boot bank-switched EPROM containing:

• an extensive supplement to BASIC for easy, custom MIDI programming
• a real-time 4-track sequencer
• a MIDI-data monitor for hex/binary/decimal real-time data display
• an interface/cable auto-test program

All programs are automatically available on power-up and can be switched out to run other manufacturers' software. The package includes: MIDI interface with EPROM installed; two MIDI cables; full documentation on disk; and MIDI BASIC program examples. This product is aimed at musicians, home hobbyists, students, repair technicians, and small recording studios, and for teaching and first-time users.

Available for $199.95 from ADPS, c/o Phil Honsinger, 86 Foxhunt Road, Waterloo, Ontario, Canada, N2K 2Z6, telephone (519) 886-6361.

New Commodore 128 Software from Abacus

Abacus, one of the largest publishers of books and software for Commodore home computers, announces three productivity software packages for the Commodore 128.

Speedterm 128 is a flexible, command driven terminal software package. It supports most modems for the C-128. In addition to the standard options found in most terminal programs, SpeedTerm supports Xmodem and Punter file transfer protocols, VT52 and VT100 terminal emulation with cursor keys, large 45K capture buffer and user-definable function keys. Suggested retail price is $59.95 (US).

TAS-128 is an enhanced version of Abacus' technical analysis system for stock market charting. Using TAS-128, the investor can automatically download indicators from DJN/RS or Warner and then build a variety of charts on the split screen: 7 moving averages, 3 oscillators, 5 volume indicators, comparison charts, trading bands, least squares and others. It incorporates many new powerful features, such as macro capabilities, automatic and unattended logon and fast draw charts using up to 4 windows. Suggested retail price is $59.95 (US).

PPM-128 is the upgraded C-128 version of Personal Portfolio Manager for tracking the performance of stocks, bonds or options. PPM-128 is a very easy to use package and has complete reporting capabilities. It also tracks profits and losses for tax purposes. The earlier C-64 version has been very favourably reviewed in the major magazines. Suggested retail price is $59.95 (US).

For more information, contact: Scott Slaughter or Jan Lloyd, Abacus Software, 2201 Kalamazoo S.E., P.O Box 7211, Grand Rapids, MI 49510 (616) 241-5510, Telex 709 101.

New Amiga Software and Books from Abacus

Abacus announces four productivity software packages for the Amiga:

TextPro is an intermediate level, high quality word processor. TextPro features fast on-screen formatting, automatic hyphenation, capability to include graphics with text, 30 user-definable function keys and flexible printer driver installation. It is designed with fast entry of text in mind. Suggested retail price is $79.95. **BeckerText** is a professional quality word processor. In addition to the standard options found in other word processors, BeckerText features fast WYSIWYG formatting, up to 999 characters per line, multiple-column printing, real time online dictionary for type-along spell checking, automatic hyphenation, decimal tab settings, numeric calculations within text, automatic index generation and more. Suggested retail price is $150 (US).

DataRetrieve is Abacus' data management package that has been fully rewritten for the Amiga. Some of its features: store and display data fields in different type styles and sizes; create and work with subsets of a file; easily change file definition and format; supports RAM disk for high speed operation. DataRetrieve also has fast search and sorting capabilities, can handle records up to 64,000 characters, allows numeric values with up to 15 significant digits, accesses up to 8 files simultaneously, indexes up to 80 different fields and has complete, built-in reporting capabilities. Suggested retail price is $79.95 (US).

AssemPro is a machine language development package, and includes an integrated editor, a high speed macro assembler with 32-bit arithmetic, large operating system library, unique debugger with 68020 single-step emulation, disassembler and reassembler. Runs from the Workbench or the CLI. Suggested retail price is $99.95 (US).

Abacus also announces four new Books for the Amiga line:

AmigaBASIC – Inside & Out is a step-by-step guide to programming in AmigaBASIC. All commands with syntax and parameters are fully described. Topics include graphics, sound, file management, more. Working programs are also included: video titling for OBJECT animation, bar and pie charts, windows and pull-down menus, using the mouse commands, sequential and random file handling, speech programming and sound synthesis. 550 pages. Suggested retail price is $24.95 (US).

Amiga Tricks and Tips is a collection of short programs for all Amiga users. Techniques include using AmigaBASIC, accessing Intuition, making the most of the CLI, DOS and the disk drive, advanced graphics programming using windows and menus, more. 275 pages. Price: $19.95 (US).

Amiga for Beginners introduces the new Amiga owner to Intuition, the mouse, CLI and AmigaBASIC. The user will learn how to use the CLI for performing many housekeeping chores and will take the first steps in BASIC programming. Price: $16.95 (US).

Amiga Machine Language describes the 68000 processor, address modes and instruction set. It details the powerful Amiga libraries for using AmigaDOS, Intuition, and speech and sound capabilities from machine language. 225 pages. Price: $19.95 (US).

Contact Julie Carle or Rob Lun at Abacus (address in above item).

Power Windows: Release 2.0

Inovatronics Inc. of Dallas presents release 2.0 of PowerWindows, an improved version of the established Commodore Amiga programmers' tool. It allows the interactive design of functional windows, gadgets, menus and entire custom screens including palette control, using the mouse and a few simple keystrokes. PowerWindows automatically generates the source code needed to integrate these constructs into original programs. This package now generates TDI Modula-2 source code as well as Manx and Lattice C and 68000 assembly language.

Available for $89.95 from: Inovatronics Inc, 11311 Stemmons Frwy., Suite 8, Dallas, Texas, 75229, telephone (214) 241-9515.

Ketek announces new Command Centre

KETEK announces a Command Centre for the Commodore 64 and 64C. As with the 128 Command Centre, this cabinet consolidates all peripherals into a compact enclosure. It keeps all cables hidden, out of sight and reach. Valuable desk space is saved enabling you to operate more efficiently, turning the ordinary desk into the ideal computer workstation. The new Command Centre is a sturdy, colour-coordinated cabinet designed to give your system a more professional appearance. The cabinet includes a main power control switch that controls the computer and all peripherals, a cooling fan to prevent overheating, and a built-in AC power strip with surge protection and line noise filtering. Other options available include a cartridge port extension and a modular telephone plug with its own on-line/off-line switch. Contact: Ketek, P.O. Box 203, Oakdale, IA, 52319 (319) 338-7123.

SpeedScript Upgrade for the C-128

SpeedPlus-128 converts your C-64 copy of SpeedScript 3.X into a full-featured 80-column C-128 version with 64K text memory and 20K erase buffer, all for use in 128 operating mode. Other enhancements include justification, tabs, two column/two side printing, word wrap toggle, selectable print-out, preview of documents, insertion of text files within a document, display of up to 26 help files, adjustable

screen display of text for increased typing speed, and more. SpeedPlus-128 is available by mail order for $29.95, including shipping and handling, from: *LIDON Enterprises, P.O. Box 773, Elm Grove, WI 53122.*

MPS-801 Descender ROM

From Public Domain Solutions comes an EPROM for the MPS-801 printer that replaces the original ROM, adding true descenders and enhancing the entire characters set to make documents look better. The EPROM comes with complete, easy to follow instructions. Price is $29.95 (US) plus $1 shipping/handling. Contact: Public Domain Solutions, P.O. Box 832, Tallevast, Florida, 34270. Orders: 1-800-634-5546; inquiries (813) 378-2394.

Synthesizer Software

Sound Quest Inc. has introduced two new Editor/Librarians for the Yamaha DX711 and the Roland D50. Both products, utilizing the Commodore Amiga, have been designed exclusively for the professional, semi-professional and serious amateur musician.

The DX11 Master Editor/Librarian provides "Musician Friendly" Help Screens for each Editor Window. It loads data files from disk into the Amiga or directly to the synthesizer either singly or in batches. As well, the Master loads stored System Exclusive data to any appropriate MIDI instrument, not just DX data files.

Multi-tasking is utilized, which permits the editing of as many data banks as desired (limited only by the computer's memory). Eight Bank Editing functions may be performed on any sized group of patches at one time. As well, the DX11 Master Editor/Librarian stores and edits all 13 types of DX711 Sys Ex data, and adds the following screens: Fractional Scaling (featuring Amiga mouse control and superior graphics), Micro-Tuning (in cents or notes), Performance, Set-Up, and Additional Parameters. In addition, the Master also provides Random Voice Generation that is personally controlled.

The D50 Master Editor/Librarian provides an identical helpful environment (data loading, bank editing, and data sending features) as the DX11. Editing, however, is performed on partials, tones, or patches from a main window and features Pop-Up windows for graphic envelope editing. The D50 Master Editor/Librarian has been designed with the capability to "Lock" any combination of partials or tones together for simultaneous editing.

For more information, contact: Glenn Hayworth, Sound Quest Inc., 5 Glenaden Avenue East, Toronto, Ontario, M8Y 2L2

Musical Catechism Lessons on the 64

"We Sing Our Faith" is a musical presentation of Basic Christian Doctrine for the C-64, based on the traditional Little Catechism of the Province of Quebec. Programmed by Religious of The Order of the Mother of God (Ordo Dei Matris), it is suitable for ages eight to sixteen, and has been tested on youngsters from the ages of eight to seventeen in their classrooms. Because music is a great memory aid, they easily retain the material; they are able to sing the verses years after leaving the class. Adults also have shown enthusiasm for this enjoyable religious instruction. It is simple to sing along, because syllables are highlighted or underlined as the music plays.

Disk 1, "The Attributes of God; the creation", contains: The Unity and Trinity of God; God and His Perfections; The Final Destiny of Man; The Creation; Our First Parents and their Fall; The Sources of Sin. Disk 2, "The Incarnation and Redemption", contains: The Promised Redeemer; The Incarnation; The Passion and Death of Our Lord; The Resurrection and Ascension; The Descent of the Holy Spirit; The Effects of the Redemption.

Each chapter contains: Questions/Verses; Lexicon after each verse, explaining words and terminology; A multiple-choice questionnaire; a beautiful on-screen religious picture to award a high score. Options: Listen to each verse any number of times; Return to start of music or to questions missed section for a second try; Study text without playing music. Price is $25 postpaid.

Also available: "Sunday Evening At Home - With The Lord" - music for singing; scripture tester/teacher; six high-resolution religious pictures. Offering requested: $10.

Order from Monastery of The Apostles, in Canada: P.O. Box 308, St.-Jovite, PQ J0T 2H0. In the U.S.: Frontier Road, Churubusco, NY 12912

Transactor For The Amiga

Keeping you in touch with the Amiga programming community

Transactor for the Amiga brings Transactor's traditional high-level technical focus to the Amiga. This is the place to turn for the kind of information programmers and serious hackers need to know. Our own editorial staff, Nick Sullivan and Chris Zamara, are already known for their Amiga programs and articles, but we're not stopping there – before the Transactor name went on an Amiga-related publication we had to be *really* ready. So we asked the top Amiga programmers, writers and other experts to write for us, and they did. What we now have is a magazine that will be a clearing house for new ideas, programs and techniques from the best Amigans in the world. And if you still don't believe we're really ready, just take a look at this sample of articles from our soon-to-be-released first issue:

- Andy Finkel, one of the authors of the Amiga's system software, tells you about "CAOS" – Amiga's *original* Disk Operating System.
- Rob Peck, author of the Amiga's ROM Kernel manuals, shows how to use the Exec Library's List-handling capabilities for more flexible programs.
- Perry Kivolowitz, president of ASDG Inc. and author of "Facc", the floppy disk accelerator, looks at ideal programming environments.
- John Toebes, director of the Software Distillery and author of Version 4.0 of the Lattice C compiler, shows the *right* way to port a program to the Amiga.
- Matt Dillon, author of well-known freely redistributable programs like his DOS Shell, explains DOS packets, with practical example programs.
- Jim Butterfield, the original Commodore programming guru, dissects an assembly-language program and explains it piece by piece.

PLUS:

Write your own "Cycle Warrior" to fight it out in Rico Mariani's arena of battling programs!

The inside facts about the Draco language compiler from its author, Chris Gray

Detailed specifications of the new "Arp.library" from its co-authors, Scott Ballantyne

Protecting yourself from THE VIRUS

An all-assembler Update command for the CLI by Bob Rakosky, author of saf-T-net

Part 1 of a series on the theory and practice of debugging, by Metascope expert Vic Wagner

A detailed look at the structure of font files and how they work, by Transactor regular Betty Clay

Reviews of the new Lattice C compiler V4.0, Power Windows 2.0, and the new Amiga 500 RAM expansion box from Byte-By-Byte.

Regular columns by Larry Phillips, Steve Ahlstrom and Don Curtis.

If you're an Amiga programmer who wants to really take hold of the machine and make it do backflips, you've just found your magazine. And if you subscribe now, you'll still be in time for our first edition. We're going to have a hard time holding onto back issues of this one, so don't miss out. Turn to the subscription form at the centre of this magazine and fill it out today!

Transactor For The Amiga. . . Flattening the learning curve.

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!