The Transactor Book of Bits and Pieces #1

For ALL Commodore Computers!

Transactor Publishing Inc.

The Transactor Book of Bits and Pieces #1

500 Steeles Avenue Milton, Ontario, L9T 3P7

Printed in Canada

Canadian Cataloguing in Publication Data

Main entry under title:
The Transactor book of bits and pieces, #1

A collection of the Bits and pieces sections from
The Transactor, v. 4-6.
Includes index.
ISBN 0-9692086-1-8

1. Commodore computers. I. The Transactor

QA76.8.C64T73 1986 004.165 C86-094352-6

The Transactor Book of Bits and Pieces #1

edited by
Karl J.H. Hildon and Chris Zamara

produced by
Karl J.H. Hildon

Zero Page

Once again we've produced a book that was necessary as a "working tool". Like the Anthology, the "Bits Book" (our abbreviated in-house title) was designed to eliminate a lot of flipping through pages and pages of magazines trying to find that one particular item that we just *know* is in there somewhere.

You'll notice the title of this book is followed by "#1". This is a deliberate ploy to eliminate any ambiguity between this Bits Book and the possibility of a second Bits Book. We chose to include all the Bits and Pieces in every magazine from Volume 4 to Volume 6 for a few reasons: first, to keep this edition a reasonable size; second, it seemed like a logical starting point and a tidy ending point; and third, since the Bits column seems to get bigger and more compact every issue, it's very possible we'll have enough material for the second Bits Book by the end of Volume 8, at which point we'll be going crazy flipping through magazine after magazine looking for that. . . well, you know.

Some seriously meaningless items were omitted, and references to magazine articles have been removed when unnecessary. Other somewhat outdated bits have been left in purely for their historical value, but otherwise they've all been printed here as they originally appeared, except for some minor editing during the proofreading stage.

They're listed in the same order, starting from Volume 4 Issue 01 through to Volume 6 Issue 06. We considered re-grouping the information but abandoned the idea for several reasons. We could have re-grouped it by machine, by subject, by BASIC versus M.L., or a number of other ways. In the end we decided that leaving the order undisturbed would be best, and that adding a comprehensive index and cross-reference would eliminate flipping through page after page of the Bits Book. Also, by maintaining the original order, the reader can progress from beginning to end on the same path as those who read the material from one issue of the magazine to the next. One can also get a feel for the time these items were originally conceived and at what point the newer machines arrived on the scene, thus maintaining a little "machine grouping". You'll find PET/CBM material near the front, C128 and Amiga info at the back, with Plus 4 and B Series somewhere in-between; VIC 20 and Commodore 64 bits are spread pretty much over the entire book.

Although some bits are written specifically for one computer, often they can be applied to the other machines, in principal if not in practice. For example, it wouldn't be difficult to port Plus 4 dazzlers to the C128.

The cross-reference I mentioned are those numbers in square brackets beside the titles of each bit. They refer to pages which contain other bits that are either directly or indirectly related. Often they refer to bits that have been updated from an earlier issue. In such cases you'll notice that the cross-references are "bi-directional". In others the cross-reference will be "uni-directional" – it depends on the subject at the outset. For example, page 117 describes a method for salvaging squashed diskettes. It references page 182 which describes salvaging scratched files. However, page 182 does not reference 117 because information on un-squashing a floppy is not necessary for un-scratching a file on a perfectly sound diskette. A bi-directional reference here would cause a waste of time. I suppose if you had to un-scratch a file on a squashed diskette the reference might have been useful, but that's reaching too far.

The same item on 117 has a reference to page 35. Page 35 references 117 because both items deal with floppy diskettes specifically, even though they're two entirely unrelated ideas. There are other areas where we've attempted to thread root subject matter such as video chips, sound, printers, keyboards, and disk access. But if you find the cross-reference isn't steering you to the right spot, try the index. Also, if for some reason you think that two items should be considered "related", please call or write – perhaps we can make the change if there is a second press run.

If any of the items raise any questions, please write us at The Transactor. As of the completion of this book, we are also available on CompuServe. Sign on and type GO CBMNET at any "!" prompt. Officially we're in the Commodore Programming and Commodore Communications Forums with plans for a third section devoted to Commodore Magazines. However, helpful advice can be found in any of the forums of CBMNET. If you haven't yet tried your hand at online networking, see the TeleColumn in The Transactor Magazine starting with Volume 7, Issue 04.

I'd like to thank everyone who has contributed to the Bits and Pieces Column over the last few years and hope to see more over the years to come. Remember, each published bit is good for a free one year subscription.

Like everything else, book projects usually take twice as long as the initial estimate – a rule that, I'm afraid, will always be "as constant as change". The first Bits Book was no exception, so without further adieu, *it's back to work on the next magazine. . .*

Karl J.H. Hildon, Editor-in-Chief
The Transactor Magazine

dedicated to Colleen J. Hildon, my mom

Contents

The "Volume" and "Issue" refers to the issue of The Transactor that these Bits and Pieces first appeared. You won't need the magazines – these are just for reference. A "•" indicates a "screen dazzler" or items published purely for their entertainment value.

The "Verifizers"

[208, 219]

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.

For this Bits and Pieces Book, The Verifizer will often not be necessary – the programs are mostly short enough that errors will be easy to spot. Also, The Verifizer was not invented until Volume 6, Issue 01. Although most of the programs shown are not "verified", the Verifizers have been included here for those that are, and for reference should we ever discontinue listing one of the versions in future magazines.

There are four versions of Verifizer on this page; one for PET/CBMs, the VIC or C64, the Plus 4, and the C128. 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	(turn it off with SYS 637)
SYS 828 to enable the C64/VIC version	(turn it off with SYS 831)
SYS 4096 to enable the Plus 4 version	(turn it off with SYS 4099)
SYS 3072,1 to enable the C128 version	(turn it off with 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 (eg. POKE 52381,0 instead of POKE 53281,0), but 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: The PET/CBM and VIC/C64 Verifizers reside 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
```

C64 and VIC–20 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
```

Plus 4 Verifizer

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

```
CB  1200 data 165, 208,  41,  15,  24, 105, 193, 141
PE  1210 data   0,  12, 165, 208,  74,  74,  74,  74
DO  1220 data  24, 105, 193, 141,   1,  12, 108, 211
BA  1230 data   0, 165, 210,  24, 101, 208, 133, 208
BG  1240 data  96
```

C128 Verifizer (40 column mode)

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

Optical Illusion

This neat little machine language program was written by Dave Berezowski at Commodore Canada. It doesn't do very much except create a rather interesting looking screen. The program will work on 40 or 80 column machines but the 80 column seemed to be the most impressive.

```
033c ldx   #$00
033e inc   $8000, x     ;vic users must subst.
0341 inx                  screen address
0342 bne   $fa
0344 inc   $033f
0347 jmp   $e455        ;for BASIC 4.0 users
0347 jmp   $e62e        ;for BASIC 2.0 users
0347 jmp   $eabf        ;for VIC-20 users
```

As you can see, the routine is interrupt driven which means you'll need to POKE the interrupt vector to get it going.

```
poke 144, 60 : poke 145, 3
```

After servicing this code, the normal interrupt routines are executed which means you'll still see the cursor. You can even edit (and RUN) BASIC while this is running, just don't try to use the cassette buffer that it lives in or whammo! Try moving the cursor around the "affected area".

Notice that the program is self modifying, a practice that is OK for small programs but should be avoided like the plague in larger ones. Self-modifying software is the worst for debugging and finding out the hard way is not fun.

Vic users could also get this going without too much difficulty (maybe even with colour?). Just substitute the PET/CBM screen start address ($8000 in the second line) with the start address of the screen in your particular Vic, one of two possibilities, $1E00 normally or $1000 with some memory expansion units. To engage it. . .

```
poke 788, 60 : poke 789, 3
```

For BASIC 4.0 users, just type in this loader. Others will need to change just the last two DATA elements and the interrupt vector POKEs.

```
10 for j = 828 to 841 : read x : poke j, x : next
20 data 162,   0, 254,   0, 128, 232, 208, 250
30 data 238,  63,   3,  76,  85, 228
```

One last note. . .don't try to include the interrupt vector POKEs in the above program. Chances are your machine will crash because before both POKEs get executed, an interrupt occurs somewhere in-between.

Selective Directory [125, 147, 195]

Ever been searching through your diskettes for a program and found yourself sifting through SEQ and REL filenames that just seem to get in the way? Or how 'bout the opposite. . .when you're looking for an SEQ or REL filename that's lost in diskettes full of programs. Well..here's a quick way around it.

```
LOAD "$0:*=PRG",8
```

When finished, LIST will display all PRG files from the directory. It would stand to reason that matching type filenames would appear for both directories if the drive number were omitted, but such is not the case. If you leave out the drive number the disk only returns filenames from the last drive used.

Mysteriously, DLOAD won't work the same way. You must use the LOAD command followed by ',8'. Any file type can be selected though. Merely substitute PRG for SEQ, REL or USR.

Another variation. . .subtitute the * for filename patterns. This has been discussed before, but now you can look for filenames that match a pattern and are also of a particular type. . .

```
LOAD "$1:B*=SEQ",8
```

. . .would load a directory of all sequential files on drive 1 that start with 'B'.

A Most Welcome Error Message? [69]

Never thought you'd see the day an error message would be pleasant, did you? Well today is the day! Just turn on your machine, hit HOME and RETURN. Too bad you can only get it when the machine is empty!

Quick File Reader [114, 173]

This three-liner will read just about any SEQ file. It's not very sophisticated but when you just want to "take a boo" at a file, it can be typed in quickly and isn't too hard to memorize. The RVS will help to spot any trailing spaces.

```
10 open 8, 8, 8, "some file"
20 input#8, a$ : ? "[r]" a$ : if st=64 then close 8 : end
30 goto 20
```

For REL files, simply change the IF statement in line 20 to:

```
if st=64 and ds=50 then. . .
```

The Dreaded Illegal Quantity [118]

Sometimes you want to read files one byte at a time. A routine much like the one above might be used, only the INPUT# would be replaced by a GET#. There's just one minor gotcha. It seems that when a byte value of zero is retrieved by GET#, the string variable slated to receive it is set to a null string, not CHR$(0).

The most common occurence of byte-by-byte reading is with PRG files from disk. Program files contain lots of these zeroes, at least one per line of BASIC (end-of-line markers). Usually a program to read the PRG file is set up like this:

```
10 open 8, 8, 8, "some prg file,p,r"
20 get#8, a$ : print a$, asc(a$) : if st=64 then close 8 : end
30 goto 20
```

The problem is that when a zero is read into A$, the ASC(function cannot cope will a null string and bombs out with ?ILLEGAL QUANTITY ERROR. The solution? You could add an extra IF statement after the GET#, for example:

```
if a$="" then a$=chr$(0)
```

. . .but that would mean an extra line for the PRINT statement and the following IF. . .rather clumsy. Keep things tidy with:

```
print a$, asc (a$ + chr$(0))
```

The ASC(function returns the ASCII value of the first character of A$. If A$ starts with a valid character, then adding CHR$(0) will make no difference. If not, then CHR$(0) will be added to the null string and a "0" will be printed rather than the dreaded illegal quantity error.

The Mysterious Extra Records [163, 183, 202]

Those of you familiar with the Relative Record system will know that the end of a relative file is flagged by the ?RECORD NOT PRESENT error, DS=50.

However, the last record used for data is not necessarily the last record of the file.

As relative files get bigger, the DOS formats additional sectors by filling them with "empty records". An empty record starts with a CHR$(255) followed by CHR$(0)'s to the end of the record which is determined by the record length. This formatting process occurs when data is written to a record that will require more disk space than has been allocated to the file so far.

Each 256 byte sector can contain 254 bytes of data (the other 2 are used by the DOS). Let's take an example record length of 127, thus 2 records fit exactly into 1 sector. Imagine that 2 complete records have already been written to the file. Upon writing a third record, the DOS must format another sector. Two empty records are written, but the first will be replaced by the data of our third record. Closing the file causes our third record and the one empty record to be stored on the diskette.

Re-opening the file is no problem, but how do we find the next available space for writing a new record? Athough our fourth record is empty, a RECORD#lf, 4 will NOT produce a ?RECORD NOT PRESENT error and the CHR$(255) could successfully be retrieved and mistaken for valid information. Therefore, we must test the first character of the record for CHR$(255). An INPUT# of this record will result in a string of length 1, so a combination of the two conditions might be appropriate. However, INPUT#ing live records of length greater than 80 will produce ?STRING TOO LONG error, so GET# must be used in combination with an ST test:

```
1000 rem *** find next available record ***
1010 record# (lf), (rn)          :rem rn = record number
1020 get#lf, a$                  :rem get 1st char
1030 if ds = 50 then return
1040 if a$ = chr$(255) and st = 64 then return
1050 rn = rn + 1 : goto 1010
```

This subroutine will search forward from wherever you set RN initially. It stops when either a ?RECORD NOT PRESENT occurs or when an empty record is found. For larger files, you might consider starting at the end of the file an work backwards, but you'll need to find the first live record and then move the record pointer one forward.

In summary, relying on RECORD NOT PRESENT is not good enough. Although it will insure an empty record every time, it will eventually leave you with wasted disk space. Often the first record of the file is used to store a "greatest record number used" variable which is updated on closing and read back on opening. Although this is probably the cleanest approach, it will only return new record numbers. Any records that have been deleted by writing a

single CHR$(255) must be found with a subroutine like above. Possibly a combination of both these techniques will produce a more efficient filing system.

Out Of Memory Error?

```
10 gosub 20
20 goto 10
```

The problem is obvious. As line 10 calls line 20, line 20 routes to line 10, and around we go again. The "return" information placed on the stack by Gosub never gets removed. As more and more piles up, eventually the stack overflows and the ?Out Of Memory Error is displayed.

FOR–NEXT loops will also do it to you. If you start a FOR–NEXT loop and jump out of the loop with GOTO, information is left on the stack waiting for the NEXT statement to come along and use it. If another loop is opened, this information get pushed farther down the stack and will probably not be removed. This is the beginning of a mess.

There are, however, a couple of built–in safeguards. If the loop is within a subroutine, a RETURN will strip off the NEXT information as well. Likewise, if you exit a loop, but use the same loop variable to open a new one, the old "NEXT" information will be removed.

Of course nobody writes programs like this but if you get the ?Out Of Memory Error and FRE(0) indicates plenty of room available, check your loops and GOSUBs.

Stack (Crackle) Pop! [10, 188]

The following SYS's will "crackle" your stack and cause a POP. Apple users will know all about the POP command. It's used to remove one level of subroutine RETURN information. However, this POP simulator removes ALL levels of subroutine returns.

A particularly useful application is within an Input Subroutine. You may have a line that detects a certain character (eg. "@") that exits the Input routine and transfers control to a Command Input routine. Further, the Command routine might test for a special character and exit to a Menu routine.

But, jumping out of subroutines with GOTO can be hazardous (as discussed in the previous item). By using one of these SYS's, the stack is all cleaned up and potential stack overflow is prevented.

```
BASIC 1.0   : sys 50568
BASIC 2.0   : sys 50583
BASIC 4.0   : sys 46610
VIC 20      : sys 50544
C 64        : sys 42352
```

No, Wait!

Fooled! The C64 and VIC 20 SYS commands to simulate a POP, don't work. At first, the C64 SYS appeared to be working, but on further testing it was FUBAR. I didn't have a VIC 20 at the time and just assumed it would be the same plus 8192, but of course this didn't work either. The BASIC 1.0, 2.0, and 4.0 SYS calls are correct.

Using the SYS's from the earlier BASICs, I compared disassemblies and found the equivalent C64 and VIC 20 ROM code lies at:

```
64: sys 42622 ($a67e)
20: sys 50814 ($c67e)
Note: difference is 8192 ($2000)
```

But these don't work either. Argh! The results appear the same as RUN/STOP – RESTORE. The problem? Although the code at these addresses is virtually identical to the earlier machines, the code that performs the SYS command is much different. Garry Kiziak of Burlington, Ontario, explained this in detail (Volume 5, Issue 02, Page 49) and offered the following short routines to replace the above SYS calls which don't work.

```
10 clear=828 : for k=clear to clear+4 : read j : poke k,j : next k
20 data 104,104,76,126,166 :rem for the c64
20 data 104,104,76,126,198 :rem for the vic 20
```

The routine is completely relocatable, so you can put it in any (safe) place that you like. Now, a SYS CLEAR will clear the stack of all RETURNs and open FOR/NEXT loops.

POP For The Commodore 64 [9, 188]

The CLEAR routine does its job just fine. However, it may also do more than you really want. There may be times when all you want to do is 'POP' the last RETURN address off the stack. The following routine will do just that on the Commodore 64.

```
10 pop=828 : for k=pop to pop+24 : read j : poke k,j : next k
20 data 104,104,169,255,133,74,32,138,163,154,201,141,240,5
30 data 162,12,76,55,164,104,104,104,104,104,96
```

The command SYS POP will remove just the last RETURN address. Any higher levels of subroutines will be unaffected until another call to SYS POP. Also, like the normal RETURN statement, any active FOR . . . NEXT loops within the subroutine will be removed by the POP. Notice that this routine is also relocatable so it can be placed in any 'safe' place.

Universal Reset [141, 158]

Instead of switching off your computer next time you want to clear out memory, etc, try this SYS. It does the same thing without that nasty power interruption.

```
sys peek(65532) + 256 * peek(65533)
```

This works not only on all Commodore machines, but any machine that uses the 6502 as its microprocessor. The 6502 pre-defines addresses 65532 ($fffc) and 65533 ($fffd) as the locations that will contain the address of the machines reset routine.

When you "power up", the 6502 does a "JMP Indirect" to $FFFC. This is known as the Hardware Reset Vector. The HRV will change from machine to machine. Use the above statement for universality; use these when you want to reset from a particular machine:

```
BASIC 1.0 : sys 64824 ($fd38)
BASIC 2.0 : sys 64721 ($fcd1)
BASIC 4.0 : sys 64790 ($fd16)
VIC 20    : sys 64802 ($fd22)
C 64      : sys 64738 ($fce2)
```

Vertical Messages [45, 80, 81, 173]

Want to print a string vertically instead of the same old horizontal way? Easy with the windowing feature of 8032 type and Fat 40 machines.

```
  10 input"[S S]some string"; ss$
  20 a$ = ss$ : gosub 1000 : print ss$
  30 goto 10
1000 print : print tab (rnd(0)*40);
5000 print "[O]";left$("[                                        ]",len(a$));"[O]";
5010 return
```

The two "Cursor Home" characters in line 10 clear any existing window. SS$ is transferred to A$ to make the subroutine more versatile.

Line 1000 can be removed. It merely TABs a random amount across the screen for the demonstration.

Line 5000 sets a window one column wide and as high as A$ is long. The routine leaves the cursor at the bottom of the window. Upon returning, SS$ is printed and neatly scrolls into place. For one column windows, the scrolling is so fast you'll never notice it.

The special characters in line 5000 are achieved using the sequence of ESCape and RVS. After typing the quote, ESC turns quote mode off, RVS enters reverse field mode, and the letter "o" (Set Top character) is pressed. However, you're still in reverse mode; hit Shift RVS to get out. Same goes for "O" (Set Bottom). Try this program:

```
100 a$ = "                                                "
110 print" S ";      : rem ↑ 48 = 's
120 a=rnd(0)*48 : i=rnd(0)*80
130 print tab(i)" o "left$(" qqqqqqqqqqqqqqqqqqqqqqqq ",a)
    " Or "left$(a$,a);      : rem ↑ 24 down's & 1 right
140 print spc(24+a/2)" s s ";: goto 120
```

Pretty useless, eh?

Escape Without Escape

The ESC (Escape) key is found on Commodore machines with business keyboards only. On some earlier machines it does absolutely nothing. On later machines with BASIC 4.0, it serves to cancel "quotes mode"; invoked when an odd number of quote keys (") have been pressed.

However, quotes mode is also invoked when an odd number of quotes are PRINTed to the screen (eg. PRINT CHR$(34);). This can be rather offensive in an Input Subroutine, especially when you don't want to disable the quote key. After the user hits the " key, any cursor keys pressed will be displayed in their reverse field or "programmed" representation. To disable this mode, the user must either hit the ESC key (if there is one) or type another quote and DELete it. What a pain.

The following POKEs will cancel quotes mode. The POKE could be placed after a test for the quote key:

```
get a$
if a$ = chr$(34) then poke. . .
```

. . .or, more simply, executed after every key press:

```
get a$ : poke. . .
```

Here are the POKEs you'll need depending on your machine:

```
BASIC 1.0  : poke 234, 0
BASIC 2.0  : poke 205, 0
BASIC 4.0  : poke 205, 0
VIC 20     : poke 212, 0
C 64       : poke 212, 0
```

If for some reason you want to turn quote mode on, just POKE the respective location with a 1.

Shift Key Detect [113, 128, 134]

On PET/CBMs, this program will detect if the Shift key is depressed.

```
100 if peek(152) then print " shift key down "
101 print " shift key up "  : goto 100
```

And YOU say, "so what?". Well, the Shift key has one advantage over other keys in that no character is entered in the keyboard buffer to interfere with your GET and INPUT commands. As part of a piece of software its uses are limited (Superscript uses the Shift key to speed up scrolling through text and output to video). But of even greater significance is during program development. Such a statement could be used to re–direct execution, test variables, change variables, or even alter machine conditions such as toggling Upper/Lower case or STOP key disabled/enabled. For example:

```
2000 print " Press 'S' to Save Record "
2010 x = rn : gosub 10000 : get a$ : if a$ = " " then 2010
2020 if a$ <> "s" then return
2030 record #8, rn
2040 print #8, rn$
2050 . . .

10000 if peek (152) then print x;
10010 if peek (152) then 10010
10020 return
```

Of course any amount of information could be transferred to subroutine 10000. By using a common variable to receive data (ie. "x"), the subroutine remains versatile and can be called from elsewhere in your program.

Line 10010 is a loop that waits for the Shift key to be released. If you hit "Shift" and nothing happens, you've probably pressed it during this line; release and try again.

This merely demonstrates a technique. It could be much more sophisticated than shown. One improvement might be the addition of cursor position store & restore subroutine calls at the beginning and end of this "pseudo-monitor". Variables could then be displayed on, say, the top or bottom line of the screen where they won't disturb other screen contents. Then the cursor would be sent back to its previous position to retain normal program appearance.

Once again, the main PEEK will depend on:

```
BASIC 1.0 : peek (516)
BASIC 2.0 : peek (152)
BASIC 4.0 : peek (152)
VIC 20    : peek (653)
C 64      : peek (653)
```

In all cases, if the above PEEK yields a zero, the Shift key is up, otherwise down.

Other possibilities for "PRINT X;" in line 10000:

```
poke 59468,  26-peek(59468) ;flip case (pet/cbm)
poke   144, 173-peek(144)   ;stop en/disable (4.0)
poke   144,  95-peek(144)   ;stop en/disable (2.0)
poke   537, 269-peek(537)   ;stop en/disable (1.0)
```

SuperPETs With Hard Disks [41, 53]

When using the SuperPET, the language disk is usually in drive 1 of your floppy. From the SuperPET menu, by simply entering the first letter of the language (ie. "a" for APL, "b" for BASIC, etc.), the system goes off to drive 1 to begin loading.

But if you have just added a Commodore 9060 or 9090 Hard Disk, you may have noticed there is no drive 1, only drive 0. Now you must enter language load commands manually. Syntax is:

```
disk/0.APL
```

Of course, "APL" could be any language by choice.

Petunia Users Beware!

Back around 1978/79, an interface for PET/CBMs was released called "The Petunia". It combines an active digital to analog converter and a video

interface for connecting to external monitors. The unit works very well on PET/CBMs, but don't use them on your VIC or 64!

The Petunia plugged on at the PET User Port. Video lines are on the top edge and the User Port lines are on the bottom. On the VIC and 64 the User Port lines are still on the bottom edge, but video signal lines are now routed to a connector all their own. Where the video out (pin 2) used to be on the PET User Port, is now +5 volts on the VIC/C64 User Port. If the Petunia "video in" is connected to +5 volts it will fry like a banana!

Now, you ask, "How does a banana fry?". . . plug your Petunia into your VIC and you'll find out!

Supermon 64 Correction [53]

In the January '82 issue of COMPUTE!, Jim Butterfield's Supermon 64 was published with one slight error that will throw a wrench into the works.

At the end of the BASIC listing are three pokes. The middle one should be POKE 45, 235 (not ,232).

Also, Dave Berezowski of Commodore Canada suggests this mod to be included at the beginning of the program:

```
poke 53281, 12
```

The poke sets the background colour to grey which looks much nicer than the blue background from power-up

The Transactor?

Since the release of our latest "new format" Transactor, we've had a lot of new interest from a lot of different people,. . . and for different reasons. But it seems that one of the questions most often asked is, "Why is it called The Transactor?".

Back in the olden days, Commodore's very first micro to hit the market, as most of you will remember, was called the **PET**; an acronym for **P**ersonal **E**lectronic **T**ransactor. Also, the word "transactor" was defined in a dictionary somewhere (that escapes my recollection) as "a vehicle or device for transferring information from one place to another". This was also the basis of our new masthead, designed to represent outward motion from a centre, but retain the familiar border that's the same shape as the stickers put on early machines.

So, influenced by these two facts, the name "The Transactor" was created. But enough nostalgic reminiscing and sentimental memorabilia, let's have some fun.

Screen Spaced [86, 87]

This short one–liner should get an award for variety of display.

```
1 c=32 : for n=1 to 41 : c=192-c : for a=0 to n :
  for b=32768+a to 34768 step n : poke b, c : next b, a, n
```

After entering RUN, the program will appear to do nothing. . . but don't hit your STOP key. . . it just needs to warm up. Once it gets going I think you'll agree. . . "not bad for a one–liner, eh?"

The program could be sped up slightly by substituting the literal numerics for variables which would be initialized on line 0. In fact, any BASIC program on any Commodore machine will run faster by using floating point variables to represent your constants (integer variables are not as fast*). When BASIC runs into a literal numeric such as "32768", it must first interpret each character and then convert it to floating point in the "Floating Point Accumulator" (see any memory map). During this, time is also spent on error checking. (* Integer variables aren't as fast because they also need to be converted to floating point) But BASIC just loves floating point variables. The value is looked up in the simple variables table, transferred into the F.P. Accumulator, and execution proceeds. Not only that, but if the same constant is used in several places, you'll save on memory, and the run-time improves considerably too.

Getting back to our one-liner. . . it's written for the 8032 so 4032, VIC-20, and 64 users will need to make changes.

```
4032     : change 34768 to 33768.
VIC-20   : change 32768 to 4096 or 7680* and
           change 33768 to 4602 or 8186*
* use second value with 8K (or more) memory expanders.
C64      : change 32768 to 1024 and
           change 34768 to 2024
```

Mind Twister / Brain Bender

This one will dazzle you! It's the handiwork of John Stoveken in Milton Ontario. In fact, don't look at it too long or your grey matter will turn three shades of purple-ish orange and ooz out your nose onto your keyboard, and make a real mess.

The program simply fills the screen with characters and then starts playing with the Upper/Lower case register at decimal 59468. By tapping any key, Upper/Lower case mode is toggled at different rates. . . 256 in all! Hit a "1" to end the routine.

The program, like some others we've presented here in Bits & Pieces, is so fast that the video beam can't keep up with the changing display, thus producing the truly weird effects.

It's designed to work on 8032's; BASIC 2.0 and Fat Forty users will have no trouble making it work, but the results will differ. VIC-20 and C64 users will, once again, need changes; two to the screen start and end addresses and another for the Upper/Lower case control register (59468 on PET/CBMs). The code is relocatable so it will work wherever you have memory.

The BASIC loader that follows is all you need to get it going (or rather all it needs to get you going). RUN it and enter:

```
SYS 20480
```

The number "**1**" shown in **bold** on line 30 is the base character used to fill the screen. For different effects, try changing this to 193, 223, 255, or your choice. You "need-to-know-what-makes-it-tick" machine code addicts will find the source code following the loader.

```
10 for j=20480 to 20539 : read x : poke j, x : next : end
20 data 169, 128, 133,   1, 169,   0, 133,   0
30 data 168, 169,   1, 145,   0, 200, 192,   0
```

```
40 data 208, 247, 230,   1, 165,   1, 201, 136
50 data 208, 239, 169,  12, 141,  76, 232, 173
60 data  76, 232,  73,   2, 141,  76, 232, 165
70 data 151, 201, 255, 240,   7, 201,  49, 208
80 data   1,  96, 198,   2, 166,   2, 202, 208
90 data 253,  76,  31,  80
```

```
        *=   $5000
        lda  #$80     ;screen start
        sta  $01
        lda  #$00
        sta  $00
        tay
loop    lda  #$01     ;base char.
sta     ($00),y
        iny
        cpy  #$00
        bne  loop
        inc  $01
        lda  $01
        cmp  #$88     ;screen full?
        bne  loop
        lda  #$0c
        sta  $e84c    ;set graphic
loop2   lda  $e84c
        eor  #$02     ;toggle to
        sta  $e84c    ;text mode
        lda  $97      ;key pressed?
        cmp  #$ff
        beq  lop      ;no, do delay
        cmp  #$31     ;yes, is it a 1?
        bne  next     ;no, change delay
        rts           ;yes, end
next    dec  $02      ;alter delay
lop     ldx  $02      ;and do one
lop1    dex
        bne  lop1
        jmp  loop2
        .end
```

Loading C64 Programs On PET/CBMs [216]

BASIC programs in PET/CBMs commonly start at $0401. But C64 BASIC programs start at memory location $0801 (remember, there is always a "zero" in the byte preceding BASIC text space). So Commodore, in their infinite

wisdom, decided that the 64 would have the capability to re-locate PET/CBM programs. They LOAD right where the 64 wants them with all chain pointers adjusted perfectly.

Not so in the reverse situation! PET/CBMs didn't know there was going to be a C64... so programs written on the 64 will LOAD into the PET at address $0801. YUK!

There are actually two solutions to this one. First, we could move the Start-of-BASIC pointer in the PET "up" to $0801. Then a zero must be placed at $0800 to keep the operating system happy. A LIST would display your program ok, but SAVE or DSAVE would reep havoc! On PET/CBMs, SAVE always starts at $0401. This would store 1K of "fore-junk" before reaching the start of the actual program. Take this file over to the 64 and it will try and load this fore-junk, leaving memory full of organized hodgepodge. The only way around it is to use the M.L.Monitor to store the program from the PET, which is also painful because we need to find the end-of-program address for the .S command.

So forget all this... here's a much easier way (thanks to James Whitewood, Milton Ontario):

1. Type: NEW
2. LOAD the C64 program into your PET/CBM. A "LIST" at this point should result in "READY."
3. Add the following line of BASIC: 0 REM
4. Now enter: poke 1026, 8
5. Delete line 0 by typing a 0 and <RETURN>
6. lastly, enter: poke 43, peek(43)-4 : clr
7. Type "LIST" and your program will be there!

A brief explanation. In Step 1, the PET thinks no program exists because the first thing in memory is an end-of-BASIC *marker* (2 consecutive zeroes).

Step 2 loads the 64 program into the PET and sets the End-Of-BASIC *pointer*. PET now thinks a program exists in memory, but the end marker is still down at $0401, so LIST will never get past this point.

In Step 3, all memory between $0401 and the End-Of-BASIC pointer (which is pointing at the end of the 64 program) is moved up to make room for "0 rem".

The forward chain pointer for Line 0 is now pointing at $0407, the end marker. The first byte of the 64 program now lies at $0807. By altering *only* the high order byte of the chain pointer (Step 4), Line 0 is "linked" to the program residing at $0807. The low order byte need not be adjusted since it will be the same.

Step 5 deletes Line 0. The PET treats this like any other delete. The space occupied by Line 0 is reclaimed by moving the Lines above down. Line 0 is effectively "squeezed out". But since Line 0 was the first Line, the new first Line will be that of the 64 program. Presto! The 64 program is right where the PET/CBM is most comfortable with it.

Step 6 adjusts the End-Of-BASIC pointer. When a line is deleted, it is assumed it will never be longer than 255 bytes so only the low order byte of the pointer is altered unless one page boundary is crossed. In this case 4 page boundaries are crossed, so we have to do the adjustment ourselves. The CLR command cleans up all the other pointers. Omitting this step would cause 4 pages of "after-junk" to be stored on a SAVE.

Wouldn't it be nice if everything were so simple?

Cheating A Syntax Error [46, 220]

If you're extremely lazy like me, you probably take any chance to make programming more effortless. For example. The RUN command is three entire key presses followed by a Return. And you don't always hit them right. How many of you have entered "RUIN", "RUB" or "RUM". Argh! Right?

Well fellow short-cutters, Wayne Garvin of Toronto has this one for us. The keys Shift and RUN/STOP do the auto LOAD and RUN sequence. Some machines want to load from disk, others from tape, but every machine, back to the original PET, has this feature.

But try this! Type a letter (eg. "k") and then hit Shift-RUN/STOP. A ?Syntax Error will result and the LOAD is ignored. However, the characters R-U-N and Return are still in the keyboard buffer. These will be honoured as if YOU entered them, and your programs begins!

Just don't hit a number key first or the LOAD command will be entered on a program line which might mutilate existing code.

This little trick is convenient. . . it means I'll see my program errors that much faster.

More Key Combos [46, 220]

If you have an 8032, 8096 or a SuperPET, you've probably used the ":" key to pause scrolling when LISTing a program or a Catalog. Try this. LIST a program and hit the ":" key; the listing stops at the bottom of the screen. Now, with your forefinger on the ":", use your pinky to press the RUN/STOP key; scrolling resumes. To pause, release RUN/STOP; to abort, release the ":".

Another I use often is "Shift, RVS, A & L". Hit Shift first, then RVS, and press the A and L keys together. This combination does a line insert. . . handy for inserting lines into programs visually. The screen will scroll down from the cursor line and you fill in the gap.

Looks Are Deceiving!

How may syntax errors can you find in this line:

```
ifgorb>tandforinthend = storun:header " disk " ,d1,ifn
```

Some are rather obvious but don't be fooled. . . other parts will run fine.

The problem is a result of "tokenizing". When you hit Return to enter a line of text, the BASIC editor begins analyzing or *parsing* the line (from left to right) looking for patterns that match pre-defined keywords in ROM. If a match is made, the editor converts the sequence into a number or "token". Tokens are used not only to save memory (ie. keywords consume only one byte) but also for speed. During execution, BASIC need only interpret a single byte instead of string of 2 to 7 characters long.

The first is probably the most difficult to spot. The sequence G OR B will correctly calculate G OR'd with B, until the spaces are removed! Now it effectively becomes GO RB. Yes, "GO" is actually defined as a keyword that doesn't do anything, unless you like putting spaces in your GOTOs (ie. GO TO). To honour the spaced out "GO TO", Commodore had to include GO in the keyword table. Otherwise "GO" would be considered a variable during execution, followed by the keyword "TO", and a ?Syntax Error would result. Such was the case in the original BASIC 1.0. All BASICs from 2.0 on incorporated this change.

Two rules here: 1. Don't use GO for a variable, and; 2. if the variable G precedes the boolean operator OR, follow G with a space or enclose it in brackets.

This next one is easy. T AND F will read as TAN DF. Since there is no open bracket for the variable DF, ?Syntax Error occurs. Once again, follow T with a space or put it in brackets.

The same thing will happen on FOR and INT until we use spaces to separate:

```
. . .t and f or in . . .
```

is how this part must read to work.

"THEN" is processed next so END will not be detected. Thus no space is needed before the variable D. However, good programmers will insert one anyways for tidiness.

After the "=" comes S TO RUN, which is of course meaningless. For this to work at all, it must be entered as: ST OR UN

Our last syntax error lies in the HEADER command. The "I" delimiter for the *disk ID* is followed by an "F". This tokenizes to "IF" and is definitely out of place. Inserting a space after the I will avoid ?Syntax Error, but "FN" is now tokenized as if it were part of a "DEF FN" sequence. Your disk ID will now be a space and the graphic character □, which has a PET ASCII value of 165, the same value as the token for "FN". Solution: A. Don't use F as the first character of a disk ID, and; 2. Don't use a pair of characters that match a keyword. These are FN, GO, IF, ON, OR, TO, or ? (PRINT shorthand).

The statement has one more problem in general that often plagues even the most experienced programmers. At first glance, the condition appears to be:

is g or b *greater than* t and f or in

Not so! The > symbol is actually operating on the variables B and T. To achieve the above test, brackets must be used to delimit the >:

if (g or b) > (t and f or in) then. . .

You may even require brackets within brackets to ensure correct order of operations. Don't be afraid to use a few extra brackets. . . the time you save later will be well worth the extra bytes!

Incompatibilityisms: C64 [81]

If you have a 1540 disk drive, you'll need a set of upgrade ROMs to make it a 1541 to make it work properly with a Commodore 64. It seems the 1540 works ok with a VIC 20, but is incompatible for program loads on a C64. Fear not though. While you're waiting for your new chip, here is a temporary fix:

```
poke 53265, 11 : load "your program", 8 : poke 53265, 27
```

The first POKE turns off the 64's screen. The 6510 uses about 25% of its processing time servicing the VIC II video chip, mainly due to the extra features (ie. sprites, etc.) which the VIC-20 doesn't have. The 1540 delivers bits at the rate of about one every 20 ms. The 64 uses up more time servicing the screen than the character is available for. Therefore, some bits are lost. With the screen off there's no problem. After the program loads, the second POKE turns video back on.

Only input is affected. File data read with INPUT# and GET# will suffer the same horrible fate. Writing data is ok though. The 1540 has no trouble keeping up with bits offered by SAVE and PRINT# because they're slowed down by the screen. With the screen off, SAVE would be much faster (maybe too fast?).

Basically, (or rather machine languagely) the 1541 ROM is 25% slower than the 1540. This means the 64 can continue with video and still service the disk. You'll notice program LOADs are a little slower, but the tradeoff is worthwhile. The 1525 Printer is also subject to this problem.

More Incompatibilityisms: Disk [151, 202]

One other note. . . diskettes formatted on the 1540, the 1541, the 2031, the 4040 (3040 in Europe) or, if there's any still around, the 2040, can all be read by from any of these models. But don't write "interchangeably". That means if you have a 1540 disk and you insert it in a 1541 (or a 4040, etc.), you can read it but don't write on it! Some say they haven't experienced any problems with this but there have also be reports of diskette clobberation. I make it a habit to have one of each format on hand so I can pick up programs from any drive. Then I copy them onto my 4040 later on.

The same is true with 8050 diskettes on the 8250 drives. One difference though. . . upon first inserting the disk in the drive, your initial access will give an error (eg. Catalog). Clear the error channel with PRINT DS$ and give the Catalog command again. You should have no further trouble until you insert another disk. But once again, don't write on them. Instead, format a new 8250 diskette and use COPY to transfer 8050 Program and Sequential files across. If you have Relative files, you'll need to use a utility to make the transfer because REL files are formatted differently on the 8250. (Transcsribe by Richard Evers, Transactor Volume 7, Issue 02, will do it)

1540/41 Command Change [200]

All Commodore disk units support the Memory-Write command. This command works like a "disk POKE" for writing data into DOS memory. On 2040, 4040, 3040, 2031, 8050, 8250, 9060 and 9090 drives the syntax is: "m-w:" (followed by an address and data, see your disk manual for details). Syntax on 1540 and 1541 drives is: "m-w" (ie. without the colon).

This command is not to be used haphazardly. Memory-Write deposits data in DOS memory that may or may not send it into never-neverland.

VIC–20 Printer Output Bug

If a program is LISTed to the 1525 printer from the VIC–20 immediately after a SAVE to tape, the 1525 will drop characters. For example:

"beige tint" might become "big ti . . .hmm, poor choice

Well, you get the point. The fix? Simple. After the SAVE, type the following:

VERIFY <Return> <RUN/STOP>

It seems that activating the VERIFY command clears the adverse condition created by SAVE. RUN/STOP aborts the VERIFY and you can now send unbugulated listings to your printer.

Alternately, you could LIST your program before SAVE. After the SAVE, a LOAD will untangle the output routines like VERIFY.

No Interlace On VIC II Chips

Some televisions on the market have what's called "interlaced CRT scan". This means that the video beam scans all the *even* rasters during one sweep, then goes back and scans all the *odd* rasters on the next sweep. Other TVs simply scan consecutive rasters.

VIC–20 video chips have a feature called "interlaced mode". To activate it:

```
poke 36864, 133
poke 36864, 5 de-activates it
```

If your picture appears to "flutter", try the above POKE. It may or may not help. Note that game cartridges from VIC–1910 up, with one exception, allow you to toggle this feature by hitting the F7 key before the game is started. The exception is "Gorf" (VIC–1923). With this cartridge, push the joystick up instead of hitting F7.

Back to the point. . . The Commodore 64 uses a new video chip called the VIC II. This chip doesn't have the interlace mode feature. Although this is not a bug, it was included in this section because it might look like one. If it happens to you, I'm afraid you're stuck. However, the 64 has pretty good video output. Chances are you won't notice it even if you have a TV with interlaced scanning.

Zenith TV Mod

On some recent Model 3 Zenith TVs there is a problem with the vertical hold synchronization. This not only affects the 20 and the 64, but virtually any device used locally to drive it (ie. other micros, video games, VCRs, etc.). The picture will, once again, appear to "flutter" because Zenith factory sets the unit to receive its vertical sync (or interlace) signal "off air". (the signal is mixed in by the station — you may have heard the buzzletters "V–I–P" used to promote this product)

The POKE discussed in the previous segment can be used to fix it, but only for VIC 20s. However, Zenith offers this more permanent fix:

Inside the Model 3 lies a yellow wire on connector 2H of module 9–152. Disconnecting this wire will force the set to generate its own internal sync signal. This might seem simple enough, but have a dealer or qualified technician do it for you. Your warranty won't be voided, nor will you notice any change during regular TV viewing (most TVs like Sony don't even use off–air sync).

Note: There is a white wire connected next to the yellow wire which should NOT be disconnected. This problem has been observed on other Zeniths and some RCAs but no specific model numbers or fixes are available at this time.

Commodore 64 Bugs Update [215, 222]

Here is a list of all known 64 bugs to date (Nov. 1982):

1. TAB and SPC

The PRINT# command cannot be followed directly by a TAB or SPC operator. To get around this, simply precede TAB or SPC with two quotes (a "literal null string"). Eg:

```
open 4, 4
print#4, "  " tab(10) " some string "
```

2. Prompt Suppress After CONT

If a program is interrupted with the RUN/STOP key, and CONT is entered to resume execution, the prompt messages generated by the operating system will no longer be suppressed. For example, if you have CONTinued a program and a dynamic LOAD occurs, ie:

```
100 load " next module " ,8
```

the prompt, "searching for next module" will be displayed on your screen. This one is really no big deal so there's no fix, although a POKE to location 19 before using CONT might do the trick.

3. Screen Editor Crash

This one was found pretty early and you may have already heard about it. Let's say you're on the 23rd, 24th, or 25th line of the screen and you type a line that's longer than 80 character but less than 120. If you now begin deleting characters, upon deleting the 80th character (DELete from column 1 of the 3rd line *around* to column 40 of the 2nd line), your machine will appear to hang.

Apparently, upon writing a space to this location, the 64 incorrectly writes information outside of the colour table. This info actually gets written to CIA 1 which is just above the colour table ($DC00). If a certain bit is set, CIA 1 will invoke an auto LOAD/RUN (as if you hit Shift RUN/STOP). This bit will be set or unset depending on the colour of your cursor. At this point, your keyboard will seem to be disabled.

Fear not! Here is the fix (thanks to Don Lekei of North Vancouver, BC). If the "9" and the "N" keys are depressed together and then released, the prompt "PRESS PLAY ON TAPE" will appear. Do so and the screen will display "OK" and go blank. Now press the RUN/STOP key and you will regain control of your 64 with no apparent ill effects. (Note for disk users with no Datasette: connecting the cassette port pins 1 and 6, the outside pins, together will have the same effect as pressing PLAY)

To avoid this potential situation altogether, simply change the cursor colour to white (usually best), purple, green, orange, brown, grey2, or bright green.

New Kernal ROM For 64 [85, 128, 142]

Commodore has been installing new Kernal ROMs in their latest production 64s. Dubbed the "Kernal 2", it fixes bug #1 from above (and maybe #2 but not #3) and also incorporates some changes. However, it's already been discontinued in lieu of a "Kernal 3" ROM coming soon. Rumour has it that Kernal 2 machines will be updated to Kernal 3, but Kernal 1 machines will have to wait. The reason for this is a change in Kernal 2 that was made too late for it to be effective. Kernal 3 will revert back to the original method used in Kernal 1.

The change in point is this. A "Screen Clear" with Kernal 1 resulted only in the screen being written with space characters. The colour table was left untouched which means any subsequent POKE to the screen would produce a character in the previous colour assigned to the location being POKEd.

Kernal 2 works the same way as the VIC 20. Clearing the screen causes the entire contents of the colour table to be written with the same value as the background colour. Now, a POKE anywhere to the screen will produce a character that is the same colour as the background. Thus it will appear to be "not there".

Had Commodore released the original Kernal 1 C64 with the VIC–20 clear screen procedure, there would be no need for a Kernal 3. Kernal 2 renders several 64 programs already in circulation inoperative after a clear screen (WordPro 3+ 64 for one). This may be a blessing in disguise though, since new bugs have been discovered since the release of Kernal 2.

Other changes in Kernal 2/3 include a slightly different operation of the Commodore Logo key when LOADing from tape. In Kernal 1, a tape LOAD would cause the tape deck to find the program and wait indefinitely for the Logo key to be pressed before proceeding. This gives you the option of aborting the LOAD by hitting RUN/STOP. With Kernal 2/3, the 64 waits 10 seconds for you to hit the Logo key once the program is found. After 10 seconds, LOAD proceeds as if you did hit Logo.

To check for Kernal 2, enter the following line:

```
print" S " : if peek(55296) = peek(53281)+240 then print " Kernal 2 "
```

Best Monitor Picture From VIC/64 [33, 72, 99, 111]

Here are the pin designations for the 5 pin Video/Audio connector on VIC 20s and C64s. The colours are those of the Radio Shack 5 pin European plug to 4 phono jack cable (Part# 42–2394).

	VIC 20	C64
1 Red	+5V @ 10ma.	Luminance
2 No Lead	Ground	Ground
3 Grey	Audeo Out	Audeo Out
4 Black	Video Low	Video.
5 White	Video Hi	SID Audio In

If you have a VIC 20 and a B&W or colour monitor, try connecting pin 4 (Video Low) or pin 5 (Video Hi) to your monitor input. Whichever gives you the best picture will obviously be the one to use.

Commodore 64 users with video monitors have a couple of options available. For colour monitors, get yourself a phono "Y" adapter from Radio Shack. To this connect Luminance (pin 1, Red) AND Video (pin 4, Black) and plug the Y–

adapter into your monitor. You should get a much sharper picture than with just Video alone.

For 64s with B&W monitors, connect just Luminance (pin 1, Red) to the monitor input. Video seems to give a "grainy" picture but Luminance comes through nice 'n' clear!

DON'T however try either of these with your VIC 20. You'll be connecting +5 volts to your monitor and you could be in for a spark show!

Kaleidoscope

This program was dug up from the depths of my cassette tape collection. The program is about 4 years old so I don't know who wrote it, although I'd be delighted to find out.

```
100 print " S " ; : c = 0
110 for j = 0 to 7 : read ch(j) : next
120 data 160, 127, 102, 64, 91, 93, 58, 32
130 sc = 32768 : cols = 80 : lines = 25
140 mx = int(co/2-1) : lc = li/co : ck = co-0.0001
150 for h = 3 to 50
160 for i = 1 to mx
170 for j = 0 to mx
180 k = i + j
190 c = ch((j*3/(i + 3) + i*h/12) and 7)
200 s1 = sc + co*int(lc*i)
210 s2 = sc + co*int(lc*k)
220 s3 = sc + co*int(lc*(ck-i))
230 s4 = sc + co*int(lc*(ck-k))
240 poke i + s2, c : poke co-i + s2, c
250 poke i + s4, c : poke co-i + s4, c
260 poke k + s3, c : poke co-k + s3, c
270 poke k + s1, c : poke co-k + s1, c
280 next j, i, h
290 goto 150
```

The program was modified to work on all Commodore machines so it doesn't make use of colour on the VIC and 64. Before RUNning, change the variables in line 130 to suit. SC is the screen start address, the other two are rather obvious.

To get colour wouldn't be hard though. All you would need is another variable, say CT for Colour Table, set equal to its start address. Then just copy lines 200 to 270 into all the "in between" line numbers (ie. 205 - 275) and substitute CT for SC, C to CL, and C1 to C4 for S1 to S4. The variable CL is a colour that would be generated randomly, or using another cryptic statement like line 190. Right now line 190 chooses a character from the CH array which is set up with the screen poke values of 8 graphics. Change these if you wish, but likewise, another array (eg. CL(0-7)) could be set up to contain the poke values of some screen colours from which a new line 195 would select. Of course if line 195 were stereotyped from line 190, each character would get the same colour in all occurences. Try changing some I's and J's around.

VIC 20 users and Commodore 64 users with Kernal 2 will need to add this line:

```
C64 : 105 poke 53281, 13
V20 : 105 poke 36879, peek(36879) and 15 or 208
```

This changes the background colour so that the pokes to the screen will show up. With the 20 (and 64 Kernal 2), a clear screen writes the whole colour nybble table with the background colour value. Thus a poke to the screen only puts a character in screen memory that's the same colour as the background, making it "invisible". This will be averted if simultaneous colour selection is added, however, changing the background colour dynamically might also be interesting.

As is, the program's about as fast as it's gonna be in BASIC. If you have a compiler and you've got nothing better to do... Finally, any dazzling new versions would be most welcome in a future issue!

4.0 Disk Append [183]

The program to follow is fairly self explanatory.

```
10 print "basic 4.0 disk append
20 print "this routine will allow a subroutine
30 print "saved as a program file on disk to be
40 print "appended to a program in memory." : print
50 print "the subroutine must begin with a line
60 print "number greater than the last line of
70 print "basic text in memory          " : print
80 print "activate with : ";
90 read ad : rem replace with ad=??? for permanent placement
100 for j=0 to 56 : read x : poke ad+j, x : next
110 print "sys" ad; chr$(34) "file name" chr$(34) ",8"
120 rem routine is fully relocatable
130 rem first data element is start address (ad)
140 rem remove 1000 if ad is set within program
1000 data 634
1010 data 169,   0, 133, 157,  32, 125, 244, 169
1020 data  96, 133, 211, 164, 209, 208,   3,  76
1030 data   0, 191,  32,  73, 244,  32, 165, 244
1040 data  32, 210, 240, 165, 211,  32,  67, 241
1050 data  32, 192, 241,  32, 192, 241,  56, 165
1060 data  42, 233,   2, 133, 251, 165,  43, 233
1070 data   0, 133, 252,  32, 140, 243,  76,  28
1080 data 244
```

Lately I've had quite a bit of use for it. For example when you want to renumber only part of a program and don't have a selective renumber utility. First you renumber the part you want renumbered, then you bring in the rest with DiskAppend.

The version above works on BASIC 4.0 only, however if there's enough demand we'll re–cut it for the others. After running it once, I tend to save a direct load copy from the MLM. This way it can be loaded back using the monitor without disturbing the contents of memory.

Crash Your Commodore 64!

A dastardly perpetration indeed for such a fine upstanding computer. But let's do it anyway (Nyah ah ah). While looking for the problem with the C64 POP SYS published last issue, I stumbled across a most interesting crash, however it seems to only work on 64s with the racing stripes:

```
poke 783, 8 : sys 42622
```

Now hit some number keys. That's it. . . keep going. Neat eh? (Neat uh? for U.S. readers). And why is the text on the screen (ie. from above) not disturbed by all the vertizontal scrolling? Hmm. Eventually it locks up completely but no harm done. Just power down, up, and you're back to normal (your machine that is). Any more out there?

C64 TV Colour Adjust [28, 99, 111]

Don Lekei of Vancouver, B.C., has this useful note for those not satisfied with the colour output of their C64 to a television set.

First power down your 64. Open the casing and on the PC board, around the general vicinity of the Return key, you'll find a metal "box" (you can't miss it). Lift the lid off this box, being careful not to wipe off that white gunk that's on the bottom of the lid and the top of that 40 pin chip. This stuff acts as a heat transfer from the chip to the lid.

Inside the box are two white nylon adjustment pots. The one on the left is the Chroma output adjust, and the one on the right is the Clock rate adjust. Turn your 64 back on and, of course, connect it to your TV. Before changing these pots, take a black felt pen and mark each one so you can return to the original positions should you get carried away. Also, you'll need some stuff on the screen to make the adjustment by. Set a black background with POKE 53281,0 and type some lines of jibberish on the screen using the colours available from

the keyboard (ie. CTRL - WHT, RED, CYN, etc.). Once that's set up you're ready for the important part.

Using a small screwdriver (preferably plastic in case it gets dropped), turn the Chroma level down (counter-clockwise) until your TV just loses the colour signal. Ideally it will "flicker" between colour and no colour. Now adjust the Clock pot so that colour is regained and stable. This will be either a clockwise or counter-clockwise turn, but all it should need it just a "tweek". Look good? Re-assemble the machine, making sure the metal lid goes on the right way, and you're done.

Of course you may not get it looking any better than it was. If so, you can always re-set both adjustments to their original positions. If your machine is still under warranty, you may want to take this note and your 64 down to your dealer and have them do it.

CRless CMD

How many of you are making sequential listings of programs to your disk drive? If you are, you probably do something like:

```
open 8, 8, 8, " 0:prog file,s,w"
cmd 8 : list
print#8 : close 8
```

This puts a listing to the disk just as if it were to the printer. You might use this file at some later time with a terminal program, etc. But have you noticed an extra carriage return always seems to creep in at the beginning of the file? If this bothers you, read on. If not, forget I mentioned it.

The CMD command works somewhat like PRINT#. Without punctuation a CR gets sent afterwards. In future, try replacing the CMD 8 with:

CMD 8,;

This suppresses that extra preceding CR. And just a reminder, choosing a file number 128 or greater causes a Line Feed to be sent after each Carriage Return.

For some reason the comma cannot be omitted. Does this suggest there may be other combinations?

Sunny Side Up! [117, 181, 182]

It seems almost criminal what they're charging for diskettes these days. The material can't cost more than about a quarter, and they probably make about a thousand every minute. So what do we do? We cleverly hack out a notch on the opposite side of the jacket, flip them over, and presto! One double sided diskette.

But consider this. A diskette always spins in one direction, even double sided disks, which are not intended to be flipped but rather for use with double head drive units. When dirt and dust particles manage to land on the disk surface, the drive spins it into the inner lining of the jacket. Usually it collects there and doesn't bother us too much. If the disk is flipped over, rotation will now be in the opposite direction. All that crud will now be released from the lining on both sides of your disk. YUK!

I hope I'm not sounding like a diskette salesman. I suppose we just have to put up with prices until enough competition brings them down. . . kind of a "fact of datalife" (ooh, poor).

Waste Space

The following is for those who like to define your variables at the beginning of a program. If you're the type that lets all your variables be defined as execution sees them, you could be in for some peculiar results.

When BASIC "sees" a variable, the variable name and information related to it gets stuffed into the simple variables table that sits immediately above your program text. Each new variable seen causes the table to grow by 7 bytes, which cannot be reclaimed short of doing a CLR. But only floating point variables make use of all 7 bytes. String variables use only the first 5, the last 2 are wasted, and integer variables use only the first 4. MicroSoft did it this way to reduce ROM code and also to make searching through the table a little quicker.

I don't suggest you actually try to use these bytes unless you're seriously strapped for memory and another 7 bytes to store some value is absolutely unaffordable. This was meant more as a refresher on the simple variables table than anything else. However if you do need these bytes, here's how to go about it.

First you MUST define some variables at the start of your listing that you will be using later on. These variables must be either string or integer type, but depending on how many you choose to pre-define will decide how much waste space you'll have to play with later. If you're using J.B.'s String Thing,

stop right here. Jim uses the last two bytes of the first variable definition, and assumes these will start with 0. Watch out for this with some other utilities too.

Moving Strings

To demonstrate the above, I borrowed part of a "function input" idea I was tossing around. . . more on that in a minute.

Line 100 defines some variables. As you can see, we'll only be able to use the waste space from A$ and B%; the third table entry, C, must not be disturbed!

To save space, the poke/peek address is calculated in a function. FN AD takes the address stored in the Start of Variables pointer and adds Y to it (equivalent of "indirect indexed" in machine code).

```
100 a$ = "" : b% = 0 : c = 3.3
110 def fn ad(y) = y + peek(42) + peek(43)*256

500 input "some function";fu$
510 a$ = fu$
520 gosub 1000
530 rem go to plot routine, etc.

1000 poke fnad(5), peek(48)                  :rem save bottom of
1010 poke fnad(6), peek(49)                  :rem strings pointer
1020 poke 48, peek(fnad(2))-1                :rem move pointer to
1030 poke 49, 2                              :rem basic input bufr
1040 poke fnad(12), peek(47)                 :rem save Top of Arrys
1050 poke 47, 1                              :rem move pointer below
1060 a$ = a$                                 :rem $0200 & rebuild a$
1070 poke 2*256 + peek(fnad(2)), 0           :rem 0 marks bufr end
1080 poke 47, peek(fnad(12))                 :rem restore Top of Arys
1090 poke 48, peek(fnad(5))                  :rem restore string
1100 poke 49, peek(fnad(6))                  :rem bot pointer
1110 a$ = ""                                 :rem null a$
1120 rem sys to tokenize routine, and/or eval expression
1130 return
```

Line 500 inputs FU$ (sorry about my choice of variables but FN$ would cause ?Syntax error, and you have to admit, it probably wouldn't be used elsewhere). FU$ is transferred to A$ because A$ must be nulled later on.

With a Y argument of 5, the result of FN AD is the effective address at which we store the Bottom of Strings pointer. To store info in B% waste space, simply pass a 12 (5 + 7, and 13, 6 + 7) to FN AD.

Now the String Move part. Lines 1020 and 1030 alter the Bottom of Strings pointer to point at the BASIC input buffer. The buffer starts at $0200 (dec 512), but the pointer is set to $0200 plus the length of A$, which we get from PEEK FN AD(2) (note: this better not be >80). Before A$ is rebuilt, the Top of Arrays Pointer must be set lower than the Bottom of Strings pointer so that garbage collection is not invoked which would ruin everything. Line 1060 merely causes A$ to be transferred to the BASIC input buffer, and line 1070 marks the end of valid buffer contents with a 0.

After mending all the assaulted pointers, again using FN AD to access values stored in waste space, A$ is set to null so that garbage collection doesn't clobber anything important down in lower memory. Continuing, the program might continue to use the contents of the input buffer. Specifically, the string could be tokenized and evaluated for use with a plotter subroutine, trigonometry program, etc.

I haven't actually tested this, but the idea was to demonstrate using waste space. For one thing, the contents of the BASIC input buffer may require a conversion to screen codes before tokenization is called. For another, the buffer itself may get clobbered by subsequent BASIC operations (ie. POKE, FN, SYS, etc.). If someone decides to tackle the "function input" utility, it may require machine code for everything past the input statement. However, I'd be most interested in ANY results!

Butterware [207]

The next three programs are from who else but Jim Butterfield. The first is String Thing 64, the second is Tapemaker for BASIC 4.0, and the third is Universal Disk Change.

String Thing 64 is the Commodore 64 version of Universal String Thing, published in Transactor 01, Volume 4. This utility will input strings from the disk up to 255 characters in length, terminating only on carriage return... handy when commas and colons in files are interfering with input. Remember, the variable that receives the input must be the first one defined in the simple variables table. Also, the file number used to open the input file must be number 1. (ie. OPEN 1, 8, etc.)

String Thing has other interesting applications. The test for CR can be changed to virtually any character. Input is also controlled by the length of the buffer string (A$ below). If lines 110 and 120 are removed, input will be throttled to 15 characters maximum (len(a$)). Location 142 stores the number of characters received from the most recent input (note lines 430, 440).

I can't be sure, but a quick look at the machine language part indicates that it will work on the VIC 20 without modification.

```
50 rem ****************************
60 rem **          string thing 64          **
70 rem **          jim butterfield          **
80 rem ****************************
90 rem input string must be first variable
100 a$ = "abcdefghijklmnopq"
110 a$ = a$ + a$ + a$ + a$ + a$
120 a$ = a$ + a$ + a$
130 rem above sets string for max (255)
200 data 160,   2, 177,  45, 153, 137,   0, 200
210 data 192,   6, 208, 246, 162,   1,  32, 198
220 data 255,  32, 228, 255, 201,  13, 240,  15
230 data 164, 142, 145, 140, 200, 132, 142, 196
240 data 139, 240,   4, 165, 144, 240, 234,  76
250 data 204, 255
260 for j = 896 to 937 : read x : ch = ch + x
270 poke j, x : next j
280 if ch<>6120 then stop 300 stop
400 open 1, 8, 2, "file"
410 rem next sys same as 'input#1,a$'
420 sys 896
425 rem l = size of input (could be 0)
430 l = peek(142)
440 print left$(a$,l)
450 if st = 0 goto 420
460 close 1
```

Tapemaker for BASIC 4.0 [109]

Tapemaker will accurately make a tape from any PRG file on disk. Now you might say, ". .but all I have to do is DLOAD the program and SAVE it on tape for the same results". Not so! Not all programs load to the start of BASIC text space. Some load to the cassette buffer and others load above the start of BASIC. Tapemaker will handle these correctly.

And there's more! Tapemaker will also make tapes of PRG files that start above address $8000. The tape routines in ROM don't allow this, even if you use the machine language monitor. Now you ask, ". .what would I possibly want to save on tape that lies above hex 8000 ?".

Tapemaker is EXTREMELY position dependent. One too many or too few spaces and whammo! So in order to publish it and eliminate any potential for entry error, we've taken the entire program and put it in DATA statements. Since you need a disk drive to use Tapemaker, the two listings that follow actually generate a new program on your disk. In fact, don't waste time

making them too pretty. . . once the listing has been entered and RUN, you can discard it. . . you won't need it again.

```
100 for j=1 to 348 : read x : ch=ch+x : next
110 if ch<>32053 then stop
120 open 8, 8, 8, "@0:tapemaker 4.0,p,w
130 restore : for j=1 to 348 : read x
140 print#8, chr$(x); : next : close 8 : end
150 data   1,   4
160 data  44,   4, 100,   0, 153,  32,  34,  84
170 data  65,  80,  69,  77,  65,  75,  69,  82
180 data  32,  32,  32,  32,  32,  32,  32,  32
190 data  32,  32,  74,  73,  77,  32,  66,  85
200 data  84,  84,  69,  82,  70,  73,  69,  76
210 data  68,  34,   0,  64,   4, 110,   0, 160
220 data  49,  58, 159,  32,  49,  44,  56,  44
230 data  49,  53,  44,  34,  73,  34,   0,  98
240 data   4, 120,   0, 160,  50,  58, 133,  34
250 data  78,  65,  77,  69,  32,  79,  70,  32
260 data  70,  73,  76,  69,  32,  79,  78,  32
270 data  68,  73,  83,  75,  34,  59,  78,  36
280 data   0, 117,   4, 130,   0, 159,  50,  44
290 data  56,  44,  51,  44,  78,  36, 170,  34
300 data  44,  80,  34,   0, 144,   4, 140,   0
310 data 132,  49,  44,  69,  44,  69,  36,  58
320 data 139,  69, 179, 177,  48, 167, 153,  69
330 data  36,  58, 137,  49,  49,  48,   0, 166
340 data   4, 150,   0, 133,  32,  34,  80,  69
350 data  84,  32,  79,  82,  32,  86,  73,  67
360 data  34,  59,  86,  36,   0, 192,   4, 160
370 data   0, 151,  56,  50,  54,  44,  49,  58
380 data 139, 198,  40,  86,  36,  41, 178,  56
390 data  48, 137,  32,  49,  56,  48,   0, 219
400 data   4, 170,   0, 151,  56,  50,  54,  44
410 data  51,  58, 139, 198,  40,  86,  36,  41
420 data 179, 177,  56,  54, 137,  32,  49,  53
430 data  48,   0, 229,   4, 180,   0, 158,  49
440 data  50,  54,  53,   0, 239,   4, 190,   0
450 data 160,  50,  58, 160,  49,   0,   0,   0
460 data 162,   2,  32, 198, 255,  32, 228, 255
470 data 133, 251, 133, 201,  32, 228, 255, 133
480 data 252, 133, 202, 169,  98, 162,   5, 133
490 data  90, 134,  91,  32, 228, 255, 160,   0
500 data 145,  90, 230,  90, 208,   2, 230,  91
510 data 230, 201, 208,   2, 230, 202, 165, 150
520 data 240, 233,  32, 204, 255, 160,   0, 132
```

```
530 data 150, 200, 132, 212, 200, 177,  42, 133
540 data 209, 200, 177,  42, 133, 218, 200, 177
550 data  42, 133, 219,  32, 149, 246,  32, 140
560 data 248, 173,  58,   3,  32,  25, 246, 169
570 data  98, 162,   5, 133, 251, 134, 252, 165
580 data  90, 133, 201, 165,  91, 133, 202,  76
590 data 206, 248
```

Now LOAD Tapemaker using the appropriate file name. A LIST will show you the BASIC part but beyond this is a whole mess of machine language. DO NOT make changes. Altering the BASIC code will shift the machine language and the SYS command will probably crash your computer. Changing the SYS command won't help either 'cause the machine code is position dependent too!

When Tapemaker is run, it will prompt for the filename of the program you wish to make a tape of, and then it asks "pet or vic?". This refers to the name you just entered, not the machine it will be loaded into. That is, if you're making a tape of a PET program, enter 'p'. Enter 'v' if the filename you entered belongs to a VIC or 64 program, OR if you want the tape to be a "direct load" file such as with machine language programs. Then simply follow the instructions to come.

Universal Disk Change [214]

Disk Change allows you to temporarily change the device number of any Commodore disk unit; 2040, 4040, 2031, 8050, 8250, 1540, and 1541.

```
100 data 12, 50, 119, 0
110 input "old device number";do
120 if do<8 or do>15 then 110
150 input "new device number";dn
160 if dn"15 then 150
200 open 15, do, 15 :rem command channel
210 a$=chr$(do+32) : b$=chr$(do+64)
220 read a : if a=0 then print "disk not recognized!" : goto310
230 print#15, "m-r"chr$(a)chr$(0) : get#15, x$ : if x$<>a$ goto 220
240 print#15, "m-r"chr$(a+1)chr$(0) : get#15, x$ : if x$<>b$ goto 220
300 print#15, "m-w"chr$(a)chr$(0)chr$(2)chr$(dn+32)chr$(dn+64)
310 close 15
```

To get your unit back to the original device number (usually 8) simply power down. Or you can send this to the command channel:

```
print#15, "u:"
```

Drive 1, Are You There? [221]

Single disk drives have one very distinguishing characteristic; there's no drive 1! If you want to test this from within a program, here's a simple procedure. Open the command channel and send an I1 (initialize drive 1). Upon reading the error channel, the error returned will either be 0 for OK, or 21 for read error. This suggests there IS a drive 1, but there may or may not be a disk mounted. If a single drive is connected, the error will be 74, drive not ready.

SuperPET Bits [14, 57]

How many bits in a SuperPET? Seems like a silly question, doesn't it. Well, it is. The answer will be sure to impress your friends though. . . we'll have it here next issue, but you may want to add it up just for fun.

Index Expressions In APL

An error in the evaluate of the individual components of an index expression can cause errors or incorrect results when using WLU microAPL with a fragmented workspace. The problem exists only with Version 1.1. The expression evaluates incorrectly when a garbage collection occurs during the evaluation. The likelihood of this happening can be reduced by forcing a garbage collect (eg. M ⌊ ⍬ WA) from time to time. The problem will never occur if the individual expressions are assigned to variables prior to performing the index operation. For example:

```
x ← a [ 1 + 1 ; 2 + 2 ]
```

changes to:

```
a1 ← 1 + 1
a2 ← 2 + 2
x ← a [ a1 ; a2 ]
```

Form Feeds and SuperPET Printer Output

Normally a SuperPET printer is dedicated to the SPET to which it is connected. It is up to the person using it to ensure that the printer is positioned to the start of a page before printing a listing. If the printer is connected to several SuperPETs through a device such as a MUPET, it can be shared by more than one person. It can even be physically distant from one or more of the micros to which it is connected, and adjusting the printer before producing a listing can be rather cumbersome. In response to requests from SPET users who wish to share printers, the following program was written:

```
010 open #2, "ffpatch,prg", output
020 loop 030   read i
040   if i=999 then quit
050   print #2, chr$(i);
060 endloop 070 close #2
080 data   5, 192,   0,  41,   0,   0
090 data 134,   8, 183,   5, 149,  15
100 data  50,  57,   5, 202, 174,  98
110 data 230,   2, 193, 130,  38,  22
120 data 204,   0,  96,  52,   6, 236
130 data 100, 189, 192, 114, 204,   0
140 data  12, 237, 228, 236, 100, 189
150 data 211, 123,  50,  98,  57,   0
160 data   0,   0,   0,   2,   0, 999
```

The program is run only once. It creates a disk file called "ffpatch" (form feed patch). The patch program is run from the SPETs main menu by entering "disk.ffpatch" when the SPET is turned on, or whenever a switch from 6502 to 6809 mode is made. When it has completed the program returns to the menu. The program applies a patch to the system routine called I3ECLOSE__ so that a FormFeed character is output whenever a printer file is closed. This will prevent printer output from more than one source from appearing on any single sheet.

Simulating a GET in PASCAL

The program below simulates a BASIC GET statement to read a character from the keyboard:

```
program main ( input , output );
 var
  c  : char;
  io : file of char;
 begin
  writeln ( 'Character?' );
  reset ( io , 'keyboard' );
  c : =chr(0);
  while (c=chr(0)) do
      read ( io , c );
  writeln ( 'Character = "',c,'"' );
 end.
```

Until a key is hit, the read statement returns a null value in the variable C. When a non-null value is returned, the while-loop is terminated and the character is displayed by the second writeln statement. The character entered is not echoed directly on the screen when it is typed.

One Line Squiggle

Squiggle was one of the very first programs written for the PET 2001 back in 1977. It didn't really do very much except draw a continuous pattern of lines on the screen. Since then a lot has been learned about Commodore Basic and Squiggle has been rendered many refinements. Even though Squiggle still doesn't do very much, it now does it in only one line of Basic.

```
100 print " Sa 1 line squiggle "
110 c$ = chr$(34)
120 print " aa type <return> 3 times
130 print " a new
140 a = 548 : b = peek(57345) : if b = 75 then a = 167
150 if b = 86 or b = 220 then a = 204
160 print " aa 1pO " right$(str$(a),3) " ,0:x = 4*
    rN(1) + 1:fOi = 1to10*rN(1):?mI(" ;
170 print c$ " aQ [Lft] " c$ " ,x,1) " c$ " Q [Lft] "
    c$ " ;:pO " right$(str$(a + 1),3) " ,1 " ;
180 print " :nE:gO1 "
190 print " run
200 print " saaaaa " ;
```

As you can see there is much more than just one line here. However, this program prints another program after some decisions are made about what type of computer is underneath it (Lines 140 & 150). It will work on any Commodore machine, although BASIC 2.0 machine users will want to ensure that A = 167 as we couldn't find a 2.0 machine to test it on. Also, it's best to enter the program using Upper/Lower case, but run it in Graphics mode. . . the program doesn't switch itself.

The program won't fit into one edit line so abbreviations are used. The mnemonic [Lft] is of course a Cursor Left character. You might LIST the program after it's entered to see just how much can actually be squeezed into one line using abbreviations.

VIC 20 and Commodore 64 users could make several additions I'm sure. A random colour changer would be an interesting mod no doubt. Once again, we'll be most pleased to see any new versions but I can't imagine them fitting onto one line anymore. . . or could they?

Late N.B.: Can anyone explain why after an extended run on a Commodore 64 the RUN/STOP key is disabled? RUN/STOP–RESTORE is also locked out. Most peculiar.

Invisible Colours [99, 111]

No, this is not a new feature. . . just a reminder that by using the Commodore Logo key instead of the CTRL key in combination with the colour keys across the top, you can obtain the other 8 colours even though they aren't shown on the key fronts. Of course this only applies to C64 users since the 20 only has 8 colours. The corresponding colours are:

CTRL 1 = Black (0)	Logo 1 = Orange (8)
CTRL 2 = White (1)	Logo 2 = Brown (9)
CTRL 3 = Red (2)	Logo 3 = Lt Red (10)
CTRL 4 = Cyan (3)	Logo 4 = Gray 1 (11)
CTRL 5 = Purple (4)	Logo 5 = Gray 2 (12)
CTRL 6 = Green (5)	Logo 6 = Lt Green (13)
CTRL 7 = Blue (6)	Logo 7 = Lt Blue (14)
CTRL 8 = Yellow (7)	Logo 8 = Gray 3 (15)

Can someone think up a "jingle" using the first letters of these colours? Some of your may know this one: Bad Boys Rape Our Young Girls But Violet Gives Willingly. It's a great way to remember your resistor colour codes. A similar one for these would become a world standard!

Miscompulations [68, 106, 107, 143, 176]

The following miscellaneous compilations for the 64 come from Howard Strasberg of Toronto, Ontario.

People always want their BASIC program to be unable to list, and unable to break in to. Well, if you start the first line with a REM followed by a shifted "L", when you type LIST, the computer will display part of line one and then a ?SYNTAX ERROR! However, like everything, you can still list it by taking out line 1 or whatever your first line may be. This is a way to prevent listing before the program is run.

To prevent listing after the program is run, you simply disable LIST in your program. This is done by:

```
poke 774, 0 : poke 775, 141
```

This will make it so that when you type LIST, you get a clear screen and a READY. Also, when STOP and RESTORE are entered, these two POKEs remain so your program still cannot be listed. In order to list your program type:

```
poke 774, 26 : poke 775, 167
```

This next one will disable STOP/RESTORE and make your listing go crazy. Load a program and:

```
poke 808, peek(808)-5
```

Now try a LIST. Obviously to return to normal:

```
poke 808, peek(808)+5
```

Another good thing to do is disable SAVE so people cannot copy your program once it is run. This is done by POKE 818,0. However, you must also disable STOP/RESTORE as this will enable SAVE. By the way, with the disable LIST Pokes, the program can be SAVEd but still not listed.

Ever got your finger worn out from banging on the same key over and over again to draw a long line or something? Well, now your problem is solved. All you have to do is POKE 650,128 and now you can hold down the keys! No more worn out fingers either!

Cathode Ray Tubing [11, 80, 81, 173]

PRINTing is a term that has grown with computers since even before the dawn of their concept. For without the need for some form of output data, what use would we have for them? But unlike the pioneers of computed data who relied mainly on bulky motor driven output, most of us have come to depend on the CRT screen for most data display. A printer enters the picture only when it becomes necessary for output to be transported, stored over some period of time, or consolidated in large quantities. Indeed these limitations of the CRT may one day be history, but at the present it is all too apparrent that the CRT is only the next step in the evolution of the printer. The concept of PRINTing has only been carried over to a new media and the full potential of the CRT has become lost in the limitations we have unnaturally imposed on ourselves from the mechanics of printers.

The next couple of routines will attempt to dissolve some conditioning that us humans have allowed. Although we can't put CRTs in our pockets (yet!), we need not treat them like printers with no moving parts. And besides, who says that printing goes left to right anyway!

```
100 a$ = " now is the time for the crt to come of age "
110 cols = 80 : c = len(a$)
120 if c and 1 then a$ = a$ + chr$(32) : c = c + 1
130 for j = 1 to c/2
140 print tab(co/2-j)mid$(a$,c/2 + 1-j,1)chr$(145)
150 print tab((co/2-1) + j)mid$(a$,c/2 + j,1)chr$(145)
160 next : print : print
170 run
```

The above centers each line output. Adjust COLS to the size of your screen. The program here merely demonstrates a technique. In actual practice, A$ would be read from disk or DATA statements and line 170 would be a RETURN from a GOSUB to this routine. Line 120 ensures that A$ contains an even number of characters. CHR$(145) is a Cursor Up and could be replaced with the quotes–mode character.

Have you tried this yet? Good. If you didn't like it, you might be impressed by reversing it. If you did, try this anyway.

```
130 for j=c/2 to 1 step -1
```

Your printer would hate you for trying this one

```
100 a$ = " now is the time for the crt to come of age "
110 c=len(a$)
120 if c and 1 then  c=c+1
130 for j=1 to c : x=c*rnd(1)+1
140 print tab(x) mid$(a$,x,1)chr$(145)
150 next : print tab(1) a$ : print : run
```

Applications? CRTubing really only comes in handy when you've got a batch of text destined for the screen and the PRINT statement just doesn't seem very intellectually stimulating. Instructions for a game fall into this category and there's no need to slow it down so the reader can keep up. Besides, it might get them to read the instructions the first time.

Combomands [21, 182, 220]

This next batch of weirdisms are key combinations that invoke commands on the Commdore 64. Essentially they throw a curve at the hardware that lies just beneath the keyboard.

This one from Craig MacIntyre of Toronto. Press SHIFT, "?", and the space bar simultaneously. It's much like hitting some erroneous keys and a Shifted RUN/STOP, except it's a lot tidier in the finger department.

These are from Darren Spruyt of Gravenhurst, Ontario. Darren wouldn't reveal the outcome to us in his letter, so we're passing on the cessation. Here's part:

". . .press the plus, minus, and the pound, and hold down. Now do it again, this time holding down the shift or Commodore key. Neat! Now, holding the left shift, press and hold the keys two, three and four. One final item: hold down the left shift and press and hold down the Q, W and E keys. Have Fun!"

This has to be the fastest, cheapest wordprocessor of all time. It has full screen editing, and can handle files easily over 400 lines. It's compatible with all CBM printers, and most others, and it's fast! The program uses no memory:

```
Basic 4/2 –   open 4, 4 : cmd 4 : poke 19, 32 : list
VIC20/C64     open 4, 4 : cmd 4 : poke 22, 35 : list
```

That POKE there in the middle makes the machine omit line numbers when LISTing. It works on the screen too. All parameters for LIST work as normal, you just won't see the line numbers.

This creates some interesting possibilities. Using one of any Basic Aid type programs, you instantly have a text editor with enough flexibility to do most jobs. To enter a line, simply type a line number followed by text. Need more lines? Use your Renumber command. And most Basic Aids offer Delete, Search, and Search with Replace. To see it on the screen, just give the POKE without OPENing, etc. Adding a line or a ?SYNTAX ERROR turns it off.

TIming The Commodore 64

While on the subject of timing, several Commodore 64 users in the U.S. and Canada have noticed a slight discrepancy in the accuracy of TI and TI$; the internal clock. So *slight* that it can add up to about 2.4 seconds per minute!

The problem stems from the factory. Commodore makes two versions of the 64 that have only one difference – the crystal oscillator that generates the system clock frequency. The crystals are different to accommodate the different types of colour TV sets. Namely, PAL sets such as those made in the U.K. and Europe, and NTSC (North American Television Transmission Standards Commission) sets like those found in North America. (C64 users elsewhere should check with a service department to see which they have)

Unfortunately, televisions are rather picky things. Without getting too much into detail, NTSC sets operate around a frequency of about 1.02273 MHz. In PAL sets the number is about 0.98525 Mhz. Although Commodore installs the proper crystal into 64s depending on destination, they don't make two different sets of ROMs to adjust the TI timer accordingly on power up which, naturally, is also affected by the clock crystal.

The crystal cannot be replaced without snafuing your video output but there are two solutions here. . . the one you pick will depend largely on how much time and money you have to waste. First, you could buy a voltage converter and a round trip ticket to anywhere in Europe for you and your 64. Now run a

TI$ dependant program against your timepiece and you'll notice the problem has gone away.

The second comes from Greg Beaumont of Toronto. Try this:

```
poke 56325, 66
```

The clock is updated during interrupts. Interrupts are controlled by Countdown Timer A in CIA 1 of the 64. The low order byte of the timer (56324) continuously counts down from $FF to 0. Each time the low order byte reaches 0, the high order byte is decremented by 1. The high order byte of the timer is also a latch. The number you POKE into the latch is the number that the high order byte begins counting down from. When the high order and low order both reach 0, an interrupt is generated on the IRQ line.

Normally, the value 64 is placed in the latch at power up. The interval at which each interrupt will occur can be calculated by:

Frequency / (256 * latch) / 60 = X

Where X = the interval in 60ths of a second. When the latch value is 64, X works out to:

1.02273 e6 / (256 * 64) / 60 = 1.0404 60ths

Omit the divide by 60, and the number of clock updates per minute comes to 62.42. This is too many. Therefore, the latch value must be increased so that Timer A must go through more countdowns in order to generate an interrupt. Using a latch value of 66 results in an interrupt interval of about 1.008 60ths of a second. This is about as close as you can get short of adjusting the clock frequency.

For those that could care less about the clock, this leaves open some other possibilities. For example, is your cursor too slow, or maybe you'd like LIST to be a little less hasty. Try POKing 56325 with values lower and higher than 66. This will cause more or less (respectively) interrupts to occur and affect the general speed of your 64 based on the number of times the interrupt routines are serviced.

DATAdjuster [72, 110, 223]

Can you say Restoreshjiblinkuhwitz? Then you should have no trouble putting these next Pokerismettes to good use. As you may have have guessed, it involves the somewhat unpopular RESTORE command.

As you all know, RESTORE adjusts the DATA pointer so that the next READ command will pick up the first element of the first DATA statement in your program. However, sometimes it would be nice if the DATA pointer could be directed to the line of your choosing. A RESTORE with an optional line number parameter would have been perfect for this. Although it wouldn't be hard to write a machine code utility to do this, these next POKEs will accomplish the same result.

Consider the Current DATA Address at decimal locations 65 & 66 in your VIC 20/C64 memory maps (62 & 63 in BASIC 4.0/2.0). This pointer will always be somewhere between the beginning and end of your BASIC text. It actually starts at the beginning of BASIC and, upon a READ command, goes forward through text until it finds a DATA statement. If it gets to the end of text without finding one, an ?Out of DATA error results.

There is another pointer that will always be somewhere within BASIC text and that is the CHRGET pointer within the CHRGET subroutine. This pointer lies at decimal 122 and 123 (119 & 120 in BASIC 4.0/2.0). This is convenient. Now all we need to do is get the CHRGET pointer as close as possible to the DATA we wish to READ next, and then transfer its contents into the Current DATA Address pointer. Like this:

```
 10 rem restore x simulator or datadjuster
 20 print "which block of data?"
 30 input "1, 2, 3, or 4"; x
 40 on x gosub 100, 200, 300, 400
 50 read a$ : print a$
 60 if a$ <> "end" then 50
 70 print : goto 30
100 poke 65, peek(122) : poke 66, peek (123) : return
101 rem 4.0/2.0 users: remember to change 65 to 62,
        66 to 63, 122 to 119, 123 to 120
110 data "blk", "wht", "red", "cyn", "end"

200 poke 65, peek(122) : poke 66, peek (123) : return
210 data "one", "two", "three", "end"

300 poke 65, peek(122) : poke 66, peek (123) : return
310 data "yuk", "blech", "brap", "end"

400 poke 65, peek(122) : poke 66, peek (123) : return
410 data "buzz", "whir", "click", "crunch", "end"
```

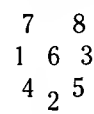
Notice that every block of DATA must be preceded by a duplicate of the adjuster subroutine. When the POKEs are finished, the Current DATA Address pointer will be somewhere just prior to the RETURN command. But that's ok. When READ is executed, BASIC goes searching for the next DATA statement.

Another variation might be to save the value of the pointer into two variables, say DL and DH, which might be pointing into the "middle" of a DATA statement. Then you could go READ some other block of data and return the pointer to the address stored in DL and DH to continue READing from the place you left off.

New 64 Video Port [71]

As mentioned in an earlier issue, Commodore is now shipping C64s with a new 8 pin video connector. Reasoning? It seems some users have been plugging their video cables into the power connector, and we can't have that.

Although there are three extra pins, only one of them has been connected to anything internally. The others don't change. The new port configuration is:



Face View

1. Luminance
2. Ground
3. Audio Out
4. Composite Video
5. Audio In
6. Chroma
7. No Connection
8. No Connection

As you can see, only Chroma has been added to the connector outputs. This will only be of use to those with monitors that have Chroma input connectors. The pattern of the connector doesn't change much either. The only incompatibility you need be concerned with is inserting 8 pin plugs into the old 5 pin ports. The only question I have is, won't the power plug now fit into the video port? If it does, and you do, watch it. Your video chip will have one more feature; air-conditioning. Colour coded stickers & tape might be a wise safety precaution under potential circumstances.

New VIC 20 Power Supply

VIC 20s are now being shipped with the same power supply as the C64. New VICs will also get the new power plug which means that they too are succeptable to the problem described above. But with a little common sense, it won't happen.

The old transformer continued to dissipate power even with the VIC turned off. This one actually shuts off with the VIC (like the 64), thus prolonging transformer life.

Three Blind Noughts [162]

With machine language becoming so popular, many "hybrid programs" are being written. A hybrid is a program combining Basic and machine language. Often the machine code begins right where the Basic text ends. But where IS the threshold point? Usually we won't need to know this, until us curious types want to look at the code that follows Basic.

Since there are no pointers set up by Basic to indicate this spot, the only way to find it is to look for the 3 consecutive zeroes that mark the end of Basic text. This is the point where the LIST function terminates. This type of work invariably requires a machine language monitor to scan memory, so if you don't have it built-in, you should arrange to load one (eg. the appropriate version of Supermon) before you load the program you wish to examine. But it's generally pretty hard to estimate the size of text in bytes from looking at the size of the LISTing, so where do you start?

The first method we all seem to try is a memory dump starting from the address where Basic text begins. As the hex listing scrolls by, we frantically scan each line hoping to catch a glimpse of three zeroes. But this can be rather tough, since they may not be all on one line. By the time empty memory rolls around, your eyes are declaring war against your brain, and of course you missed them three cursed zeroes. Feel like trying again?

Next, we could write a program to OPEN the program file on disk, read the start address, and increment it each time a byte is read until we count three zeroes in a row. Then after you calculate the address into hexadecimal, you LOAD the program, SYS to the monitor, etc, etc. But this doesn't work for tape, and besides, how many of you are actually going to have this utility handy when you need it? It would probably be faster to re-write instead. BLECH!

How 'bout using ROM routines to find it. All Commodore machines have a ROM routine that rebuilds the Basic text chain links every time a line is inserted or removed. As the routine executes, it maintains an address in RAM workspace that is continually updated until the routine reaches the end of text. Once done, this address can be viewed with the monitor.

Enter Murphy's 467th law. "A byte that contains important information will be destroyed before being allowed to examine it." The only problem with this routine is that it leaves this address in a place that the operating system uses a lot. So by the time "READY." is printed, the address is clobbered and we're no

further ahead. Therefore we need a second SYS to transfer the address to a safer spot. Here are both together, and they must be entered on the same line separated by a colon: (Note – the ":SYS4" or ":SYS8" at the end is to get you into the monitor for the .M command next, but this could be entered separately.)

```
Basic 4.0–  sys46262 : sys62792 : sys4
Basic 2.0–  sys50242 : sys62729 : sys4
VIC 20      sys50483 : sys54762 : sys8
C64         sys42291 : sys46570 : sys8
```

Enter Murphy's 468th law. "A byte you wish to transfer to a safer place will be clobbered before it gets there." This is exactly what the second SYS does. But only to the low order byte of the address. Fortunately the high order stays in tact. (Murphy doesn't have a law for addresses, just single bytes) So the second SYS (BASIC 4/2) transfers only the high order part, although it could be "backed up" to get both. The VIC20/C64 SYS's unavoidably transfer both, but again the low order is invalid.

Now, using the monitor, you can display the page number that contains the end of text marker.

```
BASIC 4/2   .m 00db 00db [or peek(219)]
VIC20/C64   .m 004f 004f [or peek(79)]
```

Although you'll still have to do some eyeballing to find the 3 blind noughts, at least you can single it down to one page of memory. If anyone has a better approach, short of writing a program, or finds a single SYS that gets the whole address, it would make a splendid update.

Retina Wrencher

If you are chronically sane and wish to stay that way, then don't, I repeat, DON'T enter this program. Of course if you know someone you'd like to see go right around the bend, just run this program for them. But don't dare look at it! Or you too will suffer the wrath of the diabolical Retina Wrencher!

The program is the handywork of Richard Evers, Toronto Ontario. It works only with machines that use the 6845 CRT Controller chip. Commonly, these are the 8032, 8096, SuperPET, and the Fat 4032.

```
10 input" [S]enter a number between 0 and 13 ";a
20 print" use the Shift key to terminate program "
30 for z=1 to 1500 : next : print" [S] "
40 poke 59520, a : z=1
```

```
50 for c=32768 to 34767 step 7
60 poke c, int(rnd(1)*255) : next
70 for d=0 to 255 step z
80 poke 59521, d
90 if peek(152) then print" ■ " : run
100 next
110 for e=255 to 0 step -z
120 poke 59521, e
130 if peek(152) then print" ■ " : run
140 z=z+1 : goto 70
```

Caution: The Zero option may cause damage to your CRT if left running for too long. Try it, but hit your Shift key right after you've had a chance to see it. To bring you this spectacular display, the zero option strains the CRT yoke somewhat. Leaving it run for too long would be like driving your car on the highway in second gear; eventually something breaks. When Shift is pressed, the screen will stay blank for a moment so don't panic. Afterwards, the characters will appear a bit smaller but will resume normal size shortly. Although The Transactor can assume no liability for damage, both Richard and I will be extremely disappointed to hear of any, especially as the result of malicious intentions. In the past, programs like this have not been released for this reason, and if there is but one report, there will be no more.

The other options are all quite harmless (to the machine that is). However, some don't do very much. If, after the initial pattern is displayed, there is no apparent activity, terminate it with Shift and try another. But once again, avoid zero, or you'll drive yourself and your machine insane.

Supermon Notes [15]

Un-beknown to many is the Supermon "P" command. The command exists in the BASIC 4.0 version as well as the VIC 20 and 64 renditions. It's used to send continuous disassembly of code to a printer. The printer must first be activated with:

```
open 4, 4 : cmd 4
```

Then give a SYS 8 (SYS 54386 on BASIC 4.0 Supermon) to get back into the monitor. Now follow "P" with Start and End Addresses and hit Return. All code between these addresses will be sent to the printer in disassembled format. Actually, any output activity that normally occurs on the screen will now be diverted to the printer. This includes "R" for Register Display, "M" for Memory Dump, and "D" for Disassemble except this will only send X number of disassembled lines where X equals the number of lines normally output on your screen. To deactivate the printer, issue a:

```
print#4 : close 4
```

As mentioned in an earlier issue, POKE 53281, 12 will change the background colour to grey offering better contrast, especially to those with C64s connected to B&W screens. But this means you first gotta exit Supermon, give the POKE, and enter the monitor again. Why not do it from Supermon? Greg Beaumonts' favourite first off is:

```
:d020 fb fc <return>
```

You might also try . . .F3 FF. . . for a fairly clear contrast or play with your own combinations.

You might be asking, "what colours are FB, FC, F3 and FF?". You could effect the same colour changes by using 0B, 0C, 03 and 0F. The leading 'F's are there mainly to avoid a '?' mark prompt when you hit return. The ? is generated when Supermon reads the location you just changed and finds it doesn't match your selection. This is because the colour registers are only 4 bits wide – the lower four bits. Remember, there are only 16 colours. . . why have 8 bits when only 4 are needed. The upper 4 are effectively non–existent and an open connection in hardware, as a rule, is always logic hi. The Fs fool Supermon into a correct verification.

Machine Code Delay [113]

Usually when we decide the only alternative is machine language it's because BASIC is just too slow. But sometimes machine code can be too fast. The following bit of code is a common subroutine for inserting delays.

```
        ldx  #$00
delay2  ldy  #$00
delay1  iny
        bne  delay1
        inx
        bne  delay2
        rts
```

As you can see there are 2 simple loops, one (using the Y register) nested within the other (using .X), and will yield about 0.25 seconds of delay on the average PET/VIC/64. But what if 1/4 second still isn't enough. You could find a countdown timer in some I/O chip to use, but this can make programs very machine dependent. JSRing to it over and over will do, but you'll need another register to increment and we've already destroyed two of them which means you'll need to use a free memory location. How 'bout a third outer loop and we'll free up the X and Y registers while we're at it.

```
        php             ;save Carry flag (optional)
delay   clc             ;clear Carry
        pha             ;# times thru inner 2 loops
        lda  #$00       ;initiate secondary loop
delay2  pha             ;save secondary loop count
        lda  #$00       ;init primary loop
delay1  adc  #$01       ;primary loop
        bcc  delay1     ;loops 255 times
        clc             ;un-set Carry
        pla             ;recall sec loop count
        adc  #$01       ;increment it
        bcc  delay2     ;255 complete?
*       pla             ;recall # iterations (C = 1)
        sbc  #$01       ;decrement
        bcs  delay      ;Carry still set?
        plp             ;restore Carry (optional)
        rts
```

With the extra code, the two inner loops will add up to a little more than .25 seconds, excluding time spent on any interrupts that may occur in the process. For fine tuning, try adjusting the second LDA operand with numbers slightly above zero. For now though, we'll assume the delay is still .25s.

Before calling the routine, the Accumulator (.A) is given a value that will be the number of iterations, plus 1, of the two inner loops. Plus 1 because the two inner loops will always execute at least once. Therefore:

Effective Delay = (.A + 1)*.25s
Minimum Delay = (0 + 1)*.25s
Maximum Delay = (255 + 1)*.25s = 64s.

The two inner loops escape to the point marked * when the Carry flag is set. Thus there is no need to do an SEC before the SBC. When the PLA at point * receives a zero, subtracting 1 results in a clear C flag and the delay ends. Otherwise, the new "outer loop" iterations value is stacked at DELAY, and the two inner loops are repeated.

Saving and restoring the P register (processor status) is optional. Since the routine is totally dependent of the Carry flag, you may wish to preserve it in case it contains "hot data" prior to your delay.

Sure, it's a little longer, but 12 bytes is a small price to pay for a more versatile solution. Plus, it will save a lot of headscratching when it's unclear whether .X and/or .Y need be preserved.

Flag Stacking

Machine language programmers are no doubt well aware of the value of the instruction "PHP"; PusH Processor status. The Status register (often called the "P" register) is, for all practical purposes, a byte stored within the microprocessor itself. This byte, like any other byte, has 8 bits. However, each bit is used to represent the occurence of some condition present in the computer as the result of a previous operation. For example, if the operation of subtracting one byte from another yields a value of zero, the bit that represents the Zero flag (or "Z" flag) is set to "1". Subsequent code might then test this flag in a decision making process for transferring execution.

Since there is only one P register, it often becomes necessary to store it for future reference while some other code is executed that may affect and change P register flags. In the previous item on delays, the routine starts with a PHP and ends with a PLP to restore the state of the Carry flag in case the code following the delay is dependent of the C flag. Remember, dependent can mean two things; dependent of Carry Clear, OR, dependent of Carry Set – a condition that cannot necessarily be assumed upon exiting the subroutine.

The same holds true for all the other flags in the P register. One that requires a particular amount of attention is the "I" flag or Interrupt Disable flag. When I is set (ie I = 1), an IRQ (Interrupt ReQuest) will be ignored by the microprocessor. Hopefully it won't be set for too long as this would disable any keyboard servicing and your machine goes into never neverland.

Let's consider the following sequence of code. It has absolutely no meaning except to demonstrate an effect:

```
. . .   ;code leading up to..
PHP     ;save P reg on stack
SEI     ;disable IRQs
. . .   ;some bunch of code. . .
PLP     ;recall P reg
```

After the "PLP", will the IRQ be accepted or ignored by the CPU? Those of you that said, "I dunno", are absolutely right! Without knowing the condition of the I flag prior to the "PHP" instruction, there is no way of giving a correct Yes or No answer. But this is good. Because no matter what it *was* it will be returned to that state by PLP, naturally! By using PHPs and complimentary PLPs, one can set or clear flags for a certain stretch of code, subroutine, etc., and return the flags to their previous state to make decisions based on previous conditions.

N.B. I wrote this piece after examining a routine that appeared to be missing a CLI instruction (CLear Interupt disable). When I took the bag off my head, I saw

why it wasn't necessary. What does seem necessary now is another article on interrupts.

Arithmetickling [63, 118]

This next item has absolutely nothing to do with your computer, but it will get you thinking. Stretch out and see if you can spot the glitch. If your skin starts crawling away, you can use your machine to iron out the wrinkles, but try it without first.

Step 1	$a = 2$
Step 2	$b = 1$
Therefore	$a = 2b$
Multiply by	$(a - b)$
Then:	$a^2 - ab = 2ab - 2b^2$
Subtract	ab
Then:	$a^2 - 2ab = ab - b^2$
Multiply by	$(a - 2b)$
Then:	$a = b$
So:	$2 = 1$
Subtract 1	$1 = 0$

But this is absurd! Uh huh, it is. I'll be leaving now.

SuperPET Bits [14, 41]

In the previous issue we posed the question, "How many bits in a SuperPET?". Did anyone try it? How many came up with the answer 1,181,104? What, you say there's more?

APL Character Set

Want to access the APL character set of your SPET? Try this:

```
poke 59520, 12 : poke 59521, 48
```

To get back, use "Escape–Reverse–n".

ACIA Status Handling

The 6551 ACIA (Asynchronous Communications Interface Adapter) is an extremely powerful and efficient chip when communication between the

SPET and other devices is required. But it has come to our attention that there is a bug in the 6551. This bug comes in the form of a discrepancy that separates the 6551 from all other 6500 series ICs.

In all other MOS interface chips such as the 6520 PIA, 6522 VIA and the new 6526 CIA, when data is received into an input register, a flag is set in another register usually known as a status register. Often this flag is "tied" to the IRQ line so that an interrupt is generated. The microprocessor then goes examining status registers to see which chip caused the interrupt, and then services it accordingly. PET, VIC, and C64 cassette tape routines work this way. When the data register is read by the CPU, the status register is cleared automatically by the internal hardware of the chip. The IC essentially prepares itself to receive and indicate the arrival of new data.

In the 6551, unlike the others, the status register is cleared not when the data register is read, but when the status register itself is read. This can be potentially hazardous during simultaneous input and output of data on your communication lines.

In programs built to handle these situations, output is usually done in the main stream of your program, for example when spooling from disk (when sending from the keyboard, characters usually only go one direction at a time and this problem will probably not occur). Input is usually handled by an interrupt routine invoked by the IRQ line. But IRQ can be generated by a number of sources. It is up to the interrupt routine to determine which source the interrupt request is coming from. The following will demonstrate how the interrupt routine could get snafued if the IRQ comes from the 6551.

```
inoutreg = $eff0
status   = $eff1
cmdreg   = $eff2
ctrlreg  = $eff3
```

Notice that the Input reg is the same as the Output reg.

In our example we'll assume that the Command and Control Registers are properly configured and that a disk file has been opened for spooling to the communications line. Typical code might look like:

```
testout lda  status    ;get status reg
        and  #16       ;last char sent?
        beq  testout
        jsr  $ffcf     ;get char from disk
        sta  inoutreg  ;put char in acia to send
        jmp  testout
```

The branching tight loop at "TESTOUT" is testing the Transmitter Data Register Empty flag, bit 4 of the Status register. When bit 4 goes high, the character has been sent and the loop is exited. Now another character can then be queued for output.

But let's say a character comes in from the line during this spooled output. In typical asynchronous communications with, for example, a host mainframe, the character could be an X-Off (transfer off) character instructing your SPET to stop sending. The 6502 always completes the current instruction before servicing an interrupt. If the X-Off were to come in during the AND or the BEQ, no problem. Your interrupt routine could examine the status reg where it would find the 6551 was the source of the interrupt. However, if the character arrives during the LDA instruction, the interrupt would still occur, but your interrupt routine would find that the flag in the 6551 status register has been cleared by LDA STATUS. This means you cannot naturally determine that an interrupt has been generated by the 6551. Since 40% of the time spent in this loop is on the LDA instruction, this could potentially occur 4 times out of 10!

There are two solutions here – one software, the other hardware. In software, your interrupt routine would examine the 6551 status register first. If this were not the source of the interrupt you would continue by testing the other chips. But if all are tested and there is still no chip claiming responsibility, then you must assume it was the 6551. This will work until you start communicating at speeds of 19,200 baud. Testing all other potential sources for generating an IRQ and then servicing the 6551 by default may not be fast enough.

The better solution is in hardware. Instead of having the 6551 generate an IRQ when characters are received, it would generate an NMI interrupt. NMI is not used for anything else in the SuperPET, so why not. Simply disconnect the 6551 from the IRQ trace and hook it up to the NMI trace. Now simply point the NMI vector at your code to handle incoming data. Since NMI doesn't do anything else, you need not even test for the source of the interrupt – just got directly to the 6551 service routine. With this modification, the bug in the 6551 will give you no further trouble.

Volume 4, Issue 05

. . .was The Reference Issue. It would become known to many as "The Brown Bible" and was the forerunner to "The Inner Space Anthology". But, there was no Bits and Pieces column in Volume 4, Issue 05.

Volume 4, Issue 06

Incrementation

Our screen dazzler this issue is "Incrementation", a concept originally from Transactor Editor, Richard Evers. Since the original version there have been several mods – portability, set-up procedures – and credit for them in line 10. Frankly, the program itself is virtually useless, but aren't all screen dazzlers? Perhaps someone looking for an eyecatcher can let us know how well it works.

Moreover, the program demonstrates once again the phenomenal speed of machine language. To start, a pattern of incrementing ASCII values is POKEd into screen memory. Hit 'S' and the machine code takes over. Quite simply, the code increments the contents of every byte in screen memory by 1, unless the byte contains a space. They are left untouched, but this part of the program could be removed for some really brain twisting effects.

The program is also a lesson in extending the boundaries of the thought process to beyond the ultimate goal. When I first saw the program in action, I immediately thought of several more difficult ways to accomplish the same effect.

Operation is simple – use the cursor keys to direct the character stream. Alternate between the space bar and cursor keys to leave gaps (ie. for messages). 'S' starts it and Shift stops it. Once stopped the pattern can be continued or cleared. The program has been published so that parts can be removed to suit your machine. Use line 15 to store changes – just space over the line# and "REM" and hit return.

Of course you need not enter the parts that will get removed, but if you do, SAVE a copy first so that it can be passed to a friend that may have different equipment than yours. For the VIC 20, the first "4" would change to a 16 or 30 depending on the high order address of your screen memory. You will also need to ensure the machine code is deposited in "real" memory which means the FOR/NEXT loop (line 120) and the SYS address (line 60) will require changes.

```
 0 rem * this version is totally relocatable !!! *
10 rem a rico mariani, chris zamara, rick evers, and karl hildon production
15 rem save" @0:incrementation",8:verify"0:incrementation",8
20 print chr$(147);
25 rem ****************************
30 rem * 4.0 - key=151: q1=196: q2=197: q3=198: sp=32
35 rem * c64 - key=203: q1=209: q2=210: q3=211: sp=60:
   poke53281,493-peek(53281)
40 rem ****************************
```

```
45 a=0: gosub120: rem * stash the data at $6000 *
50 get a$: if a$="" then 50
55 aa=asc(a$): if aa=19 or aa=147 then print a$;: goto 50
60 if a$="s" then sys24576: rem * press the 's' key to start increment
65 if peek(key)=sp then 90: rem * space bar to indicate desire to move *
70 aa=asc(a$): if aa<>17 and aa<>29 and aa<>145 and aa<>157
   then 50
75 printa$;: gosub115: pokee,a: a=a+1: if a=256 then a=0
80 if a=32 then a=33: rem it's a space !!
85 goto50
90 gosub115: pokee,peek(e)or128
95 get a$: if a$="" then 95
100 aa=asc(a$): if aa<>17 and aa<> 29 and aa<>145 and aa<>157
    then 95
105 printa$;: poke e,f: goto50
110 rem * mark the screen location and remember what was there *
115 e=peek(q1) + peek(q2)*256 + peek(q3): f=peek(e): return
120 for j=24576 to 24625: readx: poke j,x: next: return
125 rem * data for inc-80 *
130 data 169, 128, 133,   1, 169,   0, 133,   0, 168, 177,   0, 201,  32
135 data 240,  14, 133,   2, 201,  31, 208,   2, 230,   2, 230,   2, 165
140 data   2, 145,   0, 165, 152, 208,  15, 200, 192,   0, 208, 227, 230
145 data   1, 165,   1, 201, 136, 208, 219, 240, 208,  96,   0
150 rem for 4000 series change 136 in line 145 to a 132
155 rem * data for inc-commodore 64 *
160 data 169,   4, 133, 106, 169,   0, 133, 105, 168, 177, 105, 201,  32
165 data 240,  14, 133, 107, 201,  31, 208,   2, 230, 107, 230, 107, 165
170 data 107, 145, 105, 173, 141,   2, 208,  15, 200, 192,   0, 208, 226
175 data 230, 106, 165, 106, 201,   8, 208, 218, 240, 207,  96
```

Moneyout [119]

No, this program doesn't deal with the Christmas season aftermath. It's a subroutine that will format dollar figures for output. Sure you've seen lots of them before, but this 7 liner is so compact and tidy, we felt it worthy of a re-press. Of course it was written by Jim Butterfield.

The routine formats to 2 decimal places and adds trailing zeroes. V1 specifies the maximum number of digits left of the decimal place. V2 is the precision after the decimal place. An overflow will be displayed as all asterisks. The demo routine (lines 100 & 110) will show you the possibilities when the control variables are changed.

Although this routine won't dissolve any financial muck, it will make the muck look prettier.

```
100 v1 = 4: v2 = 2
110 v = rnd(1)*12000: gosub9000: printv$: run
9000 rem print format for money
9010 v4 = int(v*10↑v2 + .5)
9020 v$ = right$(" [8 spcs] " + str$(v4),v1 + v2 + 1)
9030 if v2<1 goto 9070
9040 for v5 = v1 + 2 to v1 + v2 + 1: if asc(mid$(v$,v5))<48 then nextv5
9050 v6 = v5-v1-1
9060 v$ = mid$(v$,v6,v1 + 1) + left$(" .00000 ",v6) + mid$(v$,v5)
9070 if asc(v$)>47 then v$ = left$(" ********** ",v1 + v2 + 2 + (v2 = 0))
9080 return
```

Palindrome [57, 118]

You have just become one of the few people that have actually read the word "palindrome" – it's not exactly part of everyday conversation. Nor should it be. A palindrome is something that reads the same way backwards as it does forwards. Words like mom, dad, eve, and clumsmulc (clumsmulc?) are all palindromes. Same for numbers – 23632 is a palindrome.

As it turns out, all numbers can eventually be made into palindromes, except one (more later). The idea is: pick a number, reverse the order of the digits, and add it to itself. If you don't get a palindrome, repeat the above using the result of the previous summation. For example:

```
Number:     158
Reverse     851
sum    =   1009
Reverse    9001
sum    =  10010
Reverse   01001
sum    =  11011   -  a Palindrome!
```

Not all numbers take so many iterations (eg. 56). Others require several, like 98. Regardless, Jim Butterfield felt it was a perfect candidate for a machine language program since a similar program in BASIC might take hours to eventually reach ?OVERFLOW ERROR.

The following program generates palindromes from some given value. Jim cuts it off at 255 digits – "I figure if it doesn't palindrome by 255 digits, it's not gonna. And it seems there's one relatively small number that doesn't" – What's that Jim? – "well it lies between 150 and 200." – I hate it when he does that.

```
100 data 162,   0, 142, 226,   3, 189, 219,   3
110 data  32, 210, 255, 201,  32, 240,   3, 232
120 data 208, 243,  32, 228, 255, 201,  13, 240
130 data  24, 201,  48, 144, 245, 201,  58, 176
140 data 241,  32, 210, 255,  41,  15, 174, 226
150 data   3, 157,   0,  24, 238, 226,   3, 208
160 data 225,  32, 210, 255, 234,  32, 225, 255
170 data 240, 100, 162,   0, 172, 226,   3, 169
180 data   0, 141, 227,   3, 141, 228,   3, 189
190 data   0,  24,   9,  48,  32, 210, 255,  41
200 data  15, 217, 255,  23, 240,   3, 238, 227
210 data   3,  24, 121, 255,  23, 109, 228,   3
220 data  78, 228,   3, 201,  10, 144,   5, 233
230 data  10, 238, 228,   3, 153, 255,  24, 232
240 data 136, 208, 212, 173, 228,   3, 141,   0
250 data  24, 174, 226,   3, 172, 226,   3, 173
260 data 228,   3, 240,   6, 200, 238, 226,   3
270 data 240,  20, 189, 255,  24, 153, 255,  23
280 data 136, 202, 208, 246, 169,  13,  32, 210
290 data 255, 173, 227,   3, 208, 153,  96,  86
300 data  65,  76,  85,  69,  63,  32
310 for j=828 to 993 : read x : t=t+x
320 poke j,x : next j
330 if t<>22051 then stop
400 sys 828
410 goto 400
```

Auto Liner [88]

Most program listings in print have usually been "renumbered" – they start at some line like 100 or 1000 and proceed in nice neat increments of 10. Depending on the length of the listing, just entering the line numbers can take a considerable percentage of the total time to enter the entire program. This is why several of the programmers packages available have included an Auto Line Numbering feature.

Quite simply, Auto Liners print the next line number and leave the cursor just beyond for you to enter the code. That's exactly what these do, 'cept it won't cost you anything. The first is for BASIC 4.0/2.0 users, the second for VIC 20/C64.

```
60000 input "auto: start, increment";s,i
60010 print "[Shift-Clr]"; s;:poke167,0
60020 geta$ : if a$="" then 60020
60030 print a$; : if asc(a$)<>13 then 60020
```

```
60040 p = peek(33009 + len(str$(s))) : if p = 320 or p = 160 then 60010
60050 print "s = "s + i " :i = " i " :goto60010s"
60060 poke158, 2 : poke 623,13 : poke624,13
60070 end

60000 input "auto: start, increment";s,i
60010 print "Scrq"; s;:poke204,0
60020 geta$ : if a$ = "" then 60020
60030 print a$; : if asc(a$)<>13 then 60020
60040 p = peek(1265 + len(str$(s))) : if p = 320 or p = 160 then 60010
60050 print "s = "s + i " :i = " i " :goto60010s"
60060 poke198, 2 : poke 631,13 : poke632,13
60070 end
```

DisClosed Files [126, 184]

How many times have you been in a file routine when something goes wrong. The program breaks and the disk file is left open with the LED left staring you down with a look of inadequecy. If you're just reading the file, a DCLOSE will tidy things up. That's if you have BASIC 4.0 or equivalent (ie. V/C–Link). With BASIC 2.0 you need to give the basic CLOSE command, followed by the logical file number. And if you can't remember which number you used, start looking.

If you're just reading files, cleaning up disk channels is not really critical. The disk unit can deal with this oversight and your files are not affected. But it is especially important that files open for writing are properly closed. It could mean the difference between a smooth operation and chronic teeth gnashing!

Once again, a DCLOSE command or a CLOSE followed by the correct file number will close your write–files under most conditions. But certain program errors, and especially program editing, will terminate communications with the disk, and your files are left in a state of potential doom.

Fear not! All Commodore disk units have a built–in procedure for closing any open files, read or write. Simply closing the Error Channel does it all! If the Error Channel isn't open, OPEN it – then CLOSE it. Quite simply:

```
OPEN 1, 8, 15 : CLOSE 1
```

The ", 15" specifies the Error Channel – common to all CBM drives. A look at the code in the disk ROMs shows a routine that examines all possible secondary addresses – 0 to 14 – and closes any with apparent activity. The routine is executed whenever the Error Channel is CLOSEd.

This might sound too good to be true. Well, sometimes it is. Other more nasty errors like Disk Full and machine crashes may force you to leave a file improperly closed. When this happens, an asterisk is displayed beside the file type when a Directory is printed. Do NOT Scratch them. "Star files" can be hazardous to the good health of other files. Sector links at the end of the star file might lead to other sectors that are part of another file(s). These sectors will also be de-allocated which means the disk could use them later for new stuff and whammo. Instead, use the Collect command. It will wipe out any star files while protecting the integrity of the others.

"But I've got valuable data in that blasted star file that I don't want to lose!" If that's the case you can still get at it. One little known and virtually undocumented feature of all CBM drives is the file Mishaps option. That's right - you can OPEN for Reading, Writing, or Mishaps. It allows you to dig into a star file, the result of some unfortunate event, and extract as much data as you deem necessary. For example:

```
OPEN 8, 8, 8, "0:SOME STAR FILE, S, M"
```

Actually, the M probably doesn't stand for Mishaps. Regardless, once the file is open you can start to GET# data and transfer it to a new file. Depending on how badly mangled the end of the file is, you can repair the end of the new file with direct PRINT#'s.

Once you feel the new file is complete AND properly Closed, do a Collect to purge the bad file(s).

Direct Error Reads [148]

Reading the Error Channel (Secondary Address 15) isn't a problem for BASIC 4.0 users - you simply PRINT DS$, the Disk Status string. Those without BASIC 4.0 (ie. 2.0, C64, VIC 20) are probably familiar with this subroutine:

```
60000 open 1, 8, 15
60010 input#1, e1$, e2$, e3$, e4$
60020 print e1$;e2$,e3$;e4$
60030 return
```

This routine needs to be programmed in memory because the INPUT# command won't work in direct mode. Of course if this code is already part of your program you can RUN it or GOTO it, but that can cause other headaches. The following allows reading the Error Channel in direct mode:

```
open 1, 8, 15    :rem unless already open
for j=1 to 40: sys51844 #1,e$: print e$;: if st=0 then nextj
```

This is for BASIC 2.0. For the 64 the SYS address changes to 43906 and for the VIC 20 it's 52098. Although you won't need it for this circumstance, the address for BASIC 4.0 is 48001. You can omit any spaces from the statement.

The SYS jumps into the ROM GET routine 7 bytes past the start – this is where it checks for direct mode and sends the "?illegal direct" error message. This works for GET# but not INPUT# – it checks for direct mode differently and can't be skipped short of writing your own machine language routine.

Understandably, this can become rather tedious. You might consider one of several programmers aids that include a direct mode error reading command.

Hard Disk Formatting

If you have any plans to install a Commodore hard disk drive, chances are the first command you send to it will be a New or HEADER operation. Disk users will know that this formating procedure is necessary to prepare the unit for all future operations. But once you get it started, you might as well find something else to do for a while, like learn to play piano or re–build the engine in your car. A Header operation on the hard disk can take as long as 1 hour 45 minutes because the ID you select is recorded on every sector header.

You need only do this once unless you wish to change the ID. A Header without the ID merely clears the BAM (Block Allocation Map) and the directory – the rest of the disk is left untouched. If you do decide a re–format is necessary, just remember it will take a while.

Two other hard disk notes: The unit should never be moved while the cylinder is spinning. It takes a minute or so for the cylinder to come to a complete stop after power–down. When moving it, keep the unit level – don't set it on end or its side. Hard disks should be kept on a good solid surface during operation. Even small vibrations can cause undue wear on the disk bearings. Avoid shelfs, tables with long legs or spots that may get bumped by passers–by.

Lastly, Commodore hard disks don't have a drive 0 and drive 1 – only drive 0. Some software packages assume you have a dual floppy and will attempt to access drive 1. Even BASIC tries to read drive 1 when you give a Catalog or Directory command with no drive specified. If you're experiencing any trouble, just slip in a ",D0" or "0:".

Disk De–Activity Indicator [111]

If you're disk unit gets into some long operation, like the one mentioned above, you might not notice that it's finished until the next time you browse

by. If you have a bell built into your computer, here is one way to make it useful:

```
print ds$ : poke 231, 100 : print " GGG "
```

The POKE increases the chime time of the bell, and the 3 reverse G's invoke it 3 times. If you're within ear-shot, this should be enough to get your attention. Or you can put the bell in an endless loop that stops when you hit a key. Only one problem with this though – if you happen to step out or something, that insidious chiming is enough to drive someone bonkers if left exposed to it for too long. You might come home to find your new hard disk is now a chopping block in the kitchen!

Weirdities

Here's our latest collection of crashes and assorted phenos for Commodore machines. None of them will harm your computer (unless otherwise specified) but don't try them with anything important in memory – we get enough hate mail as it is.

DLOAD'N

On any BASIC 4.0 machine type:

```
dload " [ESC][RVS] n
```

Don't ask why. We don't know. Could be like eating a power cookie.

Five and Dime

Try this on an 8000 series machine. The screen should be in upper/lower case with "unsquashed" lines. It alters the 6545 video chip and although won't hurt anything, just the same, don't let it run too long.

```
poke 59520, 5 : poke 59521, 10
```

To get out of it, hit both Shift keys and "2" (the one over the Q, not the one on the keypad).

Pirate Peeves [196, 107, 179]

Want to drive program pirates crazy? You must admit, if a burglar really wants in, he'll get in no matter how much protection you have. Program pirates are no different. The idea is to make them work for it. As they remove one lock, you check for it later in the program and throw them a couple of knuckle balls. Here's a couple of knuckle balls:

```
poke 175, 3
```

. . .switches the input device from the keyboard to the screen. For VIC 20/C64 use poke 153,3. Of course the pirate will remove this rather unsophisticated excuse for protection. So, you check for it. Then execute:

```
sys 57441
```

This turns off the keyboard completely except for carriage returns. It has no VIC20/C64 equivalent. The point is, if you make it appear as though the more they unprotect your software, the more foul it behaves, pretty soon they'll be replacing the protection they removed just so they can use it.

RAM Expansion [6]

Wanna show off to that overinflated 48K Apple user next door. Try this on 'im.

```
sys 54295  ;BASIC 4.0
```

Of course you know there can't possibly be that much, but how's he gonna know. Stay sharp though – he may know his Apple ROMs with equal impunity.

Marquis de Sade [11, 161, 179]

Try this first. It won't be too impressive to begin with but give it a chance to reach the 5 digit numbers.

```
for j=1 to 1e30 : print j " [CRSR UP] " ; : next
```

What an effective display for some 5 letter message – if I could only think of one.

Instant BASIC Monitor [96, 125, 162]

You can execute this SYS directly, but it won't mean much. However, put it on the end of some line in your program and it will report what line that is. It's part of the error message routine – the part that reports the line number after a run is interrupted. For example: ?syntax error in 6010.

Immediately upon entering direct mode, the operating system deposits an FF into the high order byte of the Current Line Number word, thus rendering that information meaningless. During program execution, the current line being executed is copied here. Try this:

```
100 print " line 100 "
110 print " some line " : sys 53112: print
120 sys 53112
130 goto 100
```

For C64 it's SYS 48585 and the VIC 20 is 56777. As you can see, if the program is renumbered, the message in line 100 will need updating. The ROM call, however, is universal. You can skip the "in" display by adding 7 to the address.

Taking it one baby step further, this routine could be used as a primitive hex to decimal converter. For example:

```
poke 55, 10 : poke 54, 7 : sys 53119
```

. . .would print the decimal equivalent of $0A07. (C64/VIC20: use 58 and 57 for the POKE addresses, with the above SYS address + 7) This is ok when you can convert the low order and high order to decimal in your head. When numbers get like CF7D, it's a little tougher. Perhaps someone with BASIC 4.0 (built-in MLM) will investigate a more sophisticated approach to the same end.

Text In Drag

Back when there were only 40 Column PET/CBMs, The Transactor discussed a method of attaching the code from one line of BASIC onto the end of the line above it. For example, the following two lines need not be on two separate lines:

```
100 for j = 1 to 10
110 next
```

The two lines would be LISTed on the screen. Then using the space bar to wrap around the top line onto the bottom line, the two lines would become one double line. Naturally it only worked for a range of 80 characters. The top line had to be less than 40 characters long and only the first 40 characters of the line below could be transfered. With the two screen lines linked together, the DELete key would be used to "drag" the line below up onto the line above, squeezing out the line number plus those extra spaces in the process. Of course you'd have to insert a colon yourself and also delete line 110. Try it! (By the way, this doesn't work on the VIC 20/C64 – the screen editor has been changed and is now too smart for this trick to be pulled off)

When 80 Column machines came along there was no longer any need for a line wrap table in low RAM – all the lines were identical regardless of how much code was entered. So the situation above could still occur, but to add one line to another required re-typing.

(You must know what I'm leading up to. Yes, a POKE this time) By making the operating system "think" that each line is longer than 80 characters, this same trick can be played on the 80 column screen editor. Slight of hand? No. More accurately, "right of hand". Location 213 is the right hand window margin. Normally it contains 79 for 80 column lines (0–79):

```
poke 213, 159
```

. . .will make the editor think that each screen line is 160 characters long, however, you're still limited to 80 characters per program line.

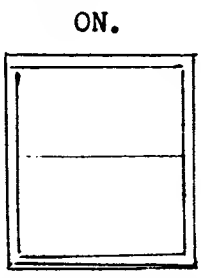
Now, about the only way I could make this work was to start from the beginning of the first line and "cursor right" all the way over to about column 75. Only then would DELete cooperate and drag the bottom line up. Using cursor left to "go around the other way" created some other problems.

Once you're done you'll want to restore 213 to a 79. Otherwise you'll get some strangeness occurring. Try POKE 213, 255 and cursor down off the bottom of the screen. Weird, eh? Again, I don't know why. And if you think that's weird, try listing a program that will cause screen scrolling. You won't crash the machine, but you may have to clear the screen before entering any new direct commands. Try experimenting. I'm not sure, but it might even work for 3 screen lines (ie. POKE 213, 239).

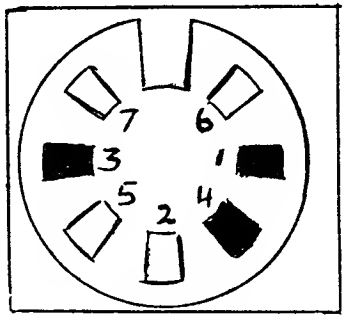
The Wooden Wedge [50]

Ted Evers of Toronto has this valuable advice: It has been observed that if the power-supply plug for the C-64 is supplied with 4-pins instead of the standard 7-pin type, damage can, and will occur to the computer and power supply circuitry if this plug is inserted into its power socket the wrong way. The shallow key of the plug will allow this to happen.

To overcome this shortcoming, fill the three unused socket holes with wooden toothpicks. The offending sockets are pin #'s 1, 3 and 4, indicated by the blocked in areas of the diagram.



C64 Right Side View



Power Plug

Some More C64 Hardware Tips [28]

In an earlier issue we published one procedure for a video allignment on your C64. Tony Lamartina of Chicago has another, but it's more suited to service technicians or those with access to necessary equipment.

1. Remove metal cover from video RF area.
2. Connect scope lead between ground and pin 1 of IC 31.
3. Adjust R27 for 1.5 volt DC level.
4. Connect scope lead between ground and pin 4 of DIN plug.
5. Adjust R25 for midposition
6. Connect decade box between pin 4 of DIN plug and right side of C78
7. Adjust resistance of decade box for 0.8 to 1 volt of signal level on scope
8. Connect resistor of value determined by decade box across pin 1 and pin 4 of DIN plug
9. Fine adjust R25 for best display of colour monitor.

Tony also suggests the following be checked on early releases of the 64:

1. Loose RF box covers. Tighten the metal tabs and re-install.
2. Check for a missing heat sink on VR2-7805 voltage regulator. Install suitable heat sink.
3. If the unit displays "sparkling" (interference across the CRT screen, random in nature) connect a 330 picoF capacitor from pin 20 (ground) to pin 30 (address 6) of the 6567 (VIC II) video chip. Make this connection outside the RF shielded box.

Octopus Syndrome

Tony also has this tip for those using a VIC20/C64 with multiple peripherals: If more than one 1541 disk drive is connected, bus lock-up will occur if all the units are cycled on at the same time (such as using a power bar main switch). To avoid this, turn on the 64 first, then one disk, then the other. Same with the serial printers – turn them on last.

DATAdjuster Update [48, 110, 223]

In Volume 4, Issue 04 we presented an item called "DATAdjuster"; a routine using POKE that would position the Data Pointer for your next READ command. It seems a potential bug can invade that version of the routine and Elizabeth Deal of Malvern, PA, has sent us a new one.

There are two POKEs that do all the work. Quite simply they deposit a copy of the CHRGET pointer into the Data Pointer. However, if a page boundary of text

memory is crossed in between the two POKEs, the high byte of the CHRGET Pointer is incremented by 1 and this value throws the Data Pointer forward by almost 256 bytes. Changing the order of the POKEs doesn't help because the same thing would happen, only in reverse – the pointer would be sent backwards through memory by 256 extra bytes. Therefore, this routine is what's called "position dependent". Of course you can't write programs to accommodate subroutines. . . quite the contrary.

Elizabeth has come up with a routine that is "position independent". It no longer uses the CHRGET Pointer but rather the BASIC CONT Pointer. This pointer is constantly updated by the operating system to point at the "link bytes" (stored at the beginning of each line of text) of the line currently being executed. The link pointer always points to the link bytes of the NEXT line of text. This value, then, is perfect for transfering to the Data Pointer. Liz subtracts 1 so that it ends up pointing at the zero byte at the end of the current line – this would be the line that contains the POKEs. The next scan for DATA would begin with the line immediately following.

Type in the demo program with no extra spaces in the first 6 lines.

```
100 data 0123456789 0123456789 0123456789
110 data 0123456789 0123456789 0123456789
120 data 0123456789 0123456789 0123456789
130 data 0123456789 0123456789 0123456789
140 data 0123456789 0123456789 0123456789
150 data 0123456789 0123456789 0123456789
160 poke 62, peek(119) : poke 63, peek(120)
170 read a$ : print a$
180 data next read should be of this line
190 data however
200 data this will never be found
210 data the colon in line 160
220 data crossed a page boundary
230 rem ----------------------
240 rem   random restore – liz deal
250 rem ----------------------
260 a1 = 58 : a2 = 62 : rem c64/vic20 = 61 & 65
270 def fn pp(q) = peek(q) + 256*peek(q + 1)
280 def fn hi(q) = int(q/256)
290 def fn lo(q) = q–256*fn hi(q)
300 y = fn pp(fn pp(a1) + 1)–1
310 poke a2, fn lo(y) : poke a2 + 1, fn hi(y)
320 read a$ : print a
330 data this is position independent
```

For VIC 20 and Commodore 64 users, line 160 changes to:

```
160 poke 65, peek(122) : poke 66, peek(123)
```

Lines 260 to 290 would be placed near the start of text – they're only executed once. Lines 300 and 310 do the work. They need to be duplicated immediately preceding each READ for which positioning is desired.

As previously discussed, one might save the contents of the Data Pointer in two other variables before altering it. This way it could be restored to its previous position at some later time.

Computenmachinen Blitzensparken

Screen dazzlers do only one thing better than tormenting a cathode ray. They demonstrate the lightning speed of machine language. The next batch of loaders are the creations of Richard Evers. Rich says, "They're really not to difficult to write, it's just that they usually turn out differently than originally planned."

The Brain (8032)

When this one was finished, it left the screen in such a state of disorder it could only be called The Brain. The code is self modifying, a no no in more serious applications, but it makes fast, compact code. The Brain checks the STOP key – we felt a good brain should at least do that.

```
2000 rem the brain 80
2010 for j=634 to 693 : read x : pokej,x : next
2020 sys 634
2030 data 169, 128, 133,  88, 169,   0, 133,  87
2040 data 168, 177,  87, 133,  89, 230,  89, 165
2050 data  89, 145,  87, 200, 208, 243, 230,  88
2060 data 165,  88, 201, 136, 208, 235, 206, 149
2070 data   2, 238, 123,   2, 173, 123,   2, 201
2080 data 132, 208, 213, 169, 128, 141, 123,   2
2090 data 169, 136, 141, 149,   2, 165, 155, 201
2100 data 239, 208, 197,  96
```

Screen Marquis (8032) [76, 161, 179]

Just like any other marquis, except this only does one screenful and it's a lot shorter. It too checks the STOP key.

```
5000 rem screen marquis 80
5010 for j=634 to 688 : read x : pokej,x : next
5020 sys 634
5030 data 169, 128, 133,  88, 169,   0, 133,  87
5040 data 160,   2, 173, 207, 135,  72, 173,   1
5050 data 128,  72, 173,   0, 128, 141,   1, 128
5060 data 177,  87, 170, 104, 145,  87, 138,  72
5070 data 200, 208, 245, 230,  88, 165,  88, 201
5080 data 136, 208, 237, 104, 104, 141,   0, 128
5090 data 165, 155, 201, 239, 208, 202,  96
```

The Boxer

This one's in BASIC, but it uses the special window commands that are only in the 8032. It also makes a great screen "set-up" program for Screen Marquis 80 or The Brain 80. Type in The Boxer first:

```
10 b=160 : c=79 : e=23
15 for d=0 to 11 : a$=chr$(d+219)
16 poke224, 0+d : poke225, 24-d : poke226, 0+d : poke213,79-d
   : print" S ";
20 for a=1 to b : print a$; : next
40 for a=1 to e
50 print" sqU " a$tab(c)a$; : next
60 b=b-4 : c=c-1 : e=e-2 : next d
70 goto 15  : rem sys 634
```

Then change line 70 from GOTO 15 to SYS 634 as shown. If you have already tried The Brain or Screen Marquis, line 70 will activate it once The Boxer is finished. Otherwise you'll have to RUN one of the previous loaders to avoid crashing.

Screen Marquis 40 [75, 161, 179]

. . .is the same as the 80 column version. The only changes are the last address of screen memory, and a short time delay loop was inserted at the end to slow it down a bit.

```
5000 rem screen marquis 40
5010 for j=634 to 698 : read x : poke j,x : next
5020 sys 634
5030 data 169, 128, 133,  88, 169,   0, 133,  87
5040 data 160,   2, 173, 231, 131,  72, 173,   1
5050 data 128,  72, 173,   0, 128, 141,   1, 128
5060 data 177,  87, 170, 104, 145,  87, 138,  72
5070 data 200, 208, 245, 230,  88, 165,  88, 201
5080 data 132, 208, 237, 104, 104, 141,   0, 128
5090 data 160, 240, 162,   0, 232, 208, 253, 200
5100 data 208, 248, 165, 155, 201, 239, 208, 192
5110 data  96
```

Commodore 64 and VIC 20 Versions

Screen Marquis for the C64 and VIC 20 is only a little more involved. The colour table must also be scrolled every time the screen is. Otherwise

characters tend to disappear whenever they are moved to a location that initially contained a space. Early 64s won't have this problem because they have a different Kernal ROM. Marquis 64 was designed to work with either Kernal.

The number shown in bold in line 5130 is the delay counter value. The lower this number, the longer the delay.

```
5000 rem marquis 64
5010 for j=828 to 924 : read x : poke j,x : next
5020 sys 828
5030 data 169,   4, 133,  88, 169,   0, 133,  87
5040 data 169, 216, 133,  91, 169,   0, 133,  90
5050 data 160,  39, 177,  87, 133,  89, 177,  90
5060 data 133,  92, 160,   0, 177,  87, 170, 177
5070 data  90, 133,  93, 165,  89, 145,  87, 165
5080 data  92, 145,  90, 134,  89, 165,  93, 133
5090 data  92, 200, 192,  40, 208, 230,  24, 165
5100 data  90, 105,  40, 133,  90,  24, 165,  87
5110 data 105,  40, 133,  87, 144, 202, 230,  91
5120 data 230,  88, 165,  88, 201,   8, 208, 192
5130 data 160, 240, 162,   0, 232, 208, 253, 200
5140 data 208, 248, 165, 145, 201, 127, 208, 160
5150 data  96
```

Marquis 20 is for unexpanded VIC 20s. The numbers in bold in lines 5030, 5040 and 5120 are, respectively, the screen start address high byte, the colour table start address high byte, and the screen end address high byte. For VICs with memory expansion that changes these addresses, change these 3 numbers to 16, 148, and 18, respectively.

Once again, the number in bold in line 5140 is the delay counter value. Make this smaller for a slower scroll.

```
5000 rem marquis vic 20
5010 for j=828 to 924 : read x : poke j,x : next
5020 sys 828
5030 data 169,  30, 133,  88, 169,   0, 133,  87
5040 data 169, 150, 133,  91, 169,   0, 133,  90
5050 data 160,  21, 177,  87, 133,  89, 177,  90
5060 data 133,  92, 160,   0, 177,  87, 170, 177
5070 data  90, 133,  93, 165,  89, 145,  87, 165
5080 data  92, 145,  90, 134,  89, 165,  93, 133
5090 data  92, 200, 192,  22, 208, 230,  24, 165
5100 data  90, 105,  22, 133,  90,  24, 165,  87
5110 data 105,  22, 133,  87, 144, 202, 230,  91
```

```
5120 data 230,  88, 165,  88, 201,  32, 208, 192
5130 data 160, 224, 162,   0, 232, 208, 253, 200
5140 data 208, 248, 165, 145, 201, 254, 208, 160
5150 data  96
```

The Brain for the C64 works the same way as the 80 column version, but is subject to the same problem as Screen Marquis. Except this time it is corrected by adjusting the background colour as opposed to the foreground colour. If you remove line 2010 before running this program, you'll notice that the spaces on your C64 screen seem to be unaffected. That's because the foreground colour of a space is the same as the background colour. The POKE in line 2010 changes the background colour to give the characters something to show up against.

```
2000 rem the brain 64
2010 poke 53281, 493-peek(53281)
2020 for j=828 to 887 : read x : pokej, x : next
2030 sys 828
2040 data 169,  4, 133,  88, 169,  0, 133,  87
2050 data 168, 177,  87, 133,  89, 230,  89, 165
2060 data  89, 145,  87, 200, 208, 243, 230,  88
2070 data 165,  88, 201,  8, 208, 235, 206,  87
2080 data   3, 238,  61,  3, 173,  61,  3, 201
2090 data   6, 208, 213, 169,  4, 141,  61,  3
2100 data 169,  8, 141,  87,  3, 165, 145, 201
2110 data 127, 208, 197,  96
```

The Plunge

The Plunge also uses the window features of the 8032 so it won't work on other Commodores.

```
4000 rem the plunge - 1984 r.t.e. the transactor
4010 for j=634 to 702 : read x : pokej,x : next
4020 sys 634
4030 data 169,  19,  32, 210, 255,  32, 210, 255
4040 data 169, 128, 133,  88, 169,  0, 133,  87
4050 data 168, 177,  87, 170, 232, 138, 145,  87
4060 data 200, 208, 246, 230,  88, 165,  88, 201
4070 data 136, 208, 238, 230, 224, 198, 225, 198
4080 data 213, 230, 226, 169, 147,  32, 210, 255
4090 data 165, 224, 201,  13, 208, 210, 165, 155
4100 data 201, 239, 208, 196, 169,  19,  32, 210
4110 data 255,  32, 210, 255,  96
```

Sequins

Sequins is another demo that shows how the cathode ray can often not keep up with the incredible speed of machine language.

```
100 rem sequins 80/40
110 for j=634 to 662 : read x : pokej,x : next
120 sys 634
130 data 162,   0, 160,   0, 254,   0, 128, 238
140 data 127,   2, 222,   0, 130, 206, 133,   2
150 data 200, 208, 241, 232, 208, 236, 165, 155
160 data 201, 239, 208, 228,  96

1000 rem sequins 64 - 1984 r.t.e. the transactor
1010 poke 53281, 493-peek(53281)
1020 for j=828 to 856 : read x : pokej, x : next
1030 sys 828
1040 data 162,   0, 160,   0, 254,   0,   4, 238
1050 data  65,   3, 222,   0,   6, 206,  71,   3
1060 data 200, 208, 241, 232, 208, 236, 165, 145
1070 data 201, 127, 208, 228,  96
```

Curtains

"Curtains" demos how the 8032 (SuperPET, and B Series) video controller chip can be altered to blank the screen. Register 6 of the 6845 controls the number of display lines on the screen. Normally it contains the number 25, naturally.

The 6845 video chip is controlled by 2 registers at 59520 and 59521 (55296 & 55297 on B Series). First 59520 is POKEd with the register number for which you want access. Then 59521 is POKEd with the value to be sent there. Both registers are write only so reading them with a PEEK will give unreliable results.

The demo also shows how text can be written to the screen while it is blank. The chip is poked with values of 1 through 25, but there's no reason why you can't go directly from 25 to 1. To stop the program, hit the SHIFT key. If you hit STOP you may find yourself left with half a screen. POKE 59521,25 will get everything back to normal.

```
100 print" S ";
110 for j=1 to 24
120 for i=1 to 79
130 print" + ";: rem fill screen
```

```
140 next i : print
150 next j
160 print" [S] "
170 poke 59520,6  : rem select reg 6
180 for j=25 to 1 step-1
190 poke 59521, j : rem write to reg 6
200 next
210 print" print on screen while blank
220 for j=1 to 25
230 poke 59521, j : rem reg 6 still selected
240 next
250 if peek(152) then poke 59521,25 : end
260 goto 180
```

Graphic Print [11, 45, 81, 173]

This next routine is rather useless the way it stands, but the part that plots the bar chart is simple and fast. The variable HT (height) could be replaced by data READ from a DATA statement.

```
10 sc=4448 : ln=22  : rem vic 20
20 sc=8032 : ln=22  : rem vic 20 w/exp
30 sc=1824 : ln=40  : rem c64
40 sc=33408 : ln=40 : rem 40 column
50 sc=34048 : ln=80 : rem 80 column
60 input " [S]enter a word ";a$
70 rem poke 53281, 12 : rem for c64
80 for i=1 to len(a$)
90 ht=asc(mid$(a$,i,1)) : lt=sc+i-ln/2*(ht-64)
100 y=sc+i : poke y+ln, ht+64
110 for j=y to lt step -ln
120 if j=lt then poke j,123 : goto 140
130 poke j, 97
140 next j, i
```

Modulo Counter [96]

Paul Obeda of London, Ontario, uses this handy little counter that will go from 1 to any value of your choosing, and then repeat, without any IF/THEN statements and independent of any FOR/NEXT loops.

```
c = -c * (c < max) + 1
```

The statement (C < MAX) will yield a result of –1 when true and 0 for false. For example, if MAX = 12 and C starts at zero, –C will be multiplied by –1 since 0 is

definitely less than 12. 1 is added and C now equals 1. Then -1 is multiplied by -1, plus 1 equals 2. And so on until C equals 12. -12 * (12<12) equals 0, plus 1 and we're back to the start.

Of course MAX could be any value, but so can +1. This could be replaced by any expression your imagination can conjure.

Reverse RVS [11, 45, 80, 173]

Setting Reverse character print is as easy as printing a RVS field control character. But there's another way that is somewhat uncommon but can be handy in the right circumstances. Suppose you want every second character of a message to be in Reverse field. Can you imagine all those control characters? Try this:

```
10 a$ = "some string"
20 for j = 1 to len(a$)
30 c = 1-c : poke 199, c
40 print mid$(a$,j,1);
50 next
60 print"[Q]" : rem cursor up
70 goto 20
```

Line 30 POKEs 199 with alternating values of 1 and 0. (Use 199 for C64/VIC 20 and 159 for BASIC 2/4) This is the RVS field flag for the operating system. When you print a RVS field control character, the OS does virtually the same thing. Location 199/159 is checked by the PRINT routine as it outputs characters.

One Line PET Emulator

Need to RUN some PET software on your Commodore 64? Try these POKEs courtesy of Jim Butterfield. Most of what they do is set the screen to $8000 (32768) and the boundaries of BASIC text space from $0400 to $8000. This will handle most programs, even those that POKE to the screen. But programs with SYS calls to machine language routines that perhaps rely on the operating system will give you trouble no matter what adjustments are made. C64-Link users will have to use the Link Relocator first.

```
1 poke56576,5: poke53272,4: poke648,128: poke1024,0
  : poke44,4: poke56,128: print"[S]" : new
```

On Error Goto [103, 133, 198]

This tidy little error trapping routine for the C64 is another Butterfield original. Jim POKEs the code in at $CF00 where it's out of the way, but it's totally relocatable so it could be set up anywhere. Line 50 adjusts the Error Message Link to point at $CF00 (207*256+0).

First, the code tests for an error. If there isn't one, it jumps to warm start or READY. If there is one, the error number is stored in $030D (781) which is also used as the X register save for SYS. (The error number is held in the X register). Then the number 1000 is placed at $14,15 and the routine jumps to the GOTO routine in ROM. Note: the stack pointer is reset to $FA so all RETURNs and FOR/NEXT loops will be popped off the stack.

Line "1000" is determined by the two numbers shown in bold on line 15 (1000 = 232 + 3*256). If you want to use a different range of lines for your error trap, just change these two numbers accordingly. For example, line 50000 would be 80 + 195*256.

```
10 data  16,   3,  76, 139, 227, 142,  13,   3
15 data 169, 232, 133,  20, 169,   3, 133,  21
20 data 162, 250, 154, 169, 167,  72, 169, 233
25 data  72,  76, 163, 168
30 for j = 52992 to 53019 : read x
40 poke j, x : next j
50 poke 768, 0 : poke 769, 207
100 rem test program
110 stop
1000 x = peek(781)
1010 if x = 2  then print " you already opened that file, numskull " : end
1020 if x = 20 then print " you can't divide by zero, calculus breath " : end
1030 if x = 11 then print " type it right this time, ninny " : end
1040 print  " something else went wrong, probably your fault "
1050 end
```

Once installed, try executing OPEN 4,4 twice, PRINT 1/0, and any old syntax error. With this routine one can write a more informative and user-friendly error status reporter. Some errors could even be fixed for the user followed by re-entry to the program.

But Seriously Folks. . . [83, 91, 111, 146, 208]

Coming up with new discoveries on your computer might be personally rewarding and intellectually stimulating, but there's no immediate recognition. Now it's time for the other extreme – a fanfare for everything you try.

Thanks to our Rick Evers, everytime you hit return with this little routine installed, you'll get a drum roll and cymbal finale.

The routine is linked in by the Input Vector of the 8032 type machines. Line 110 re-points the vector at this code, and it in turn transfers execution to the input routine.

```
100 for j=634 to 686 : read x : poke j,x : next
110 poke 233, 122 : poke 234, 2
120 data   8,  72, 138,  72, 152,  72, 169,  16
130 data 141,  75, 232, 169,  55, 141,  74, 232
140 data 169,   0, 133,   0, 141,  72, 232, 160
150 data   0, 200, 192,  21, 208, 251, 230,   0
160 data 165,   0, 201,   0, 208, 238, 141,  75
170 data 232, 141,  74, 232, 104, 168, 104, 170
180 data 104,  40,  76,  29, 225
```

Zoundz [82, 91, 111, 146, 208]

This next sound effect for the C64 is from Howard Strasberg of Toronto, Ontario. Notice how little code is required to keep the SID making sounds once it's set up properly.

```
10 s=54272
20 for l=0 to 24 : poke s+l, 0 : next
30 poke s+3, 8
40 poke s+5, 128 : poke s+6, 8
50 poke s+14, 117
60 poke s+18, 16
70 poke s+24, 143
80 for fr=1 to 24000 step 100
90 gosub 150
100 next fr
110 for fr=24000 to 1 step -100
120 gosub 150
130 next fr
140 run
150 poke s+4, 65
160 for t=1 to 4
170 fq=fr+peek(s+27)/2
180 hf=int(fq/256) : lf=fq and 255
190 poke s, lf : poke s+1, hf
200 next t
210 poke s+4, 64
220 return
```

aMAZEing quickies

This next couple of short snorts come from Chris Zamara of Downsview, Ont.

The below will work on any commodore machine, but make sure you're in upper case/graphics mode. Try this tiny program:

```
10 print mid$("/\",rnd(1)*2+1,1); : goto 10
20 rem a shifted "n" and a shifted "m"
```

Or for a different effect:

```
10 print mid$("/\X",rnd(1)*3+1,1); : goto 10
20 rem the X is a shifted "v"
```

Here's a neat variable one.

```
10 get a$ : v=val(a$) : if v then m=v*2
20 if rnd(1)<.5 then print left$("////////////////////",rnd(1)*m); : goto 10
                        : rem 18 shifted "n"'s
30 print left$("\\\\\\\\\\\\\\\\\\\\",rnd(1)*m);:goto10
40      rem 18 shifted "m"'s
```

After running the above program, press one of the number keys 1 to 9.

CBM 4032 V2.2 Disable

For those of you with 80 column machines, you may or may not know about Chuan Chee's CBM 4032 program. It converts the 8032 to behave just like a 4032 making life a little easier when you want to run a 40 column program. Version 2.2 is the latest. Early versions had minor bugs and loaded in about 10 seconds. V2.2 is clean as a whistle and boots up in a split second.

If this utility gets more than a little machine time on your computer, you need not reset the entire machine to disable it.

```
sys 14 * 4096
```

. . .disables CBM 4032 with no nasty side effects on the program in memory. Once back to 80 columns though, you'll have to re-boot to get back to 40.

Kernal 3 For The Commodore 64 [27, 142]

Commodore has released Kernal 3 – a new retro fit ROM for the C64. The "Kernal" is one of 4 ROMs found inside the 64. It's called the Kernal because it handles the fundamental or "inner most" operations of the machine. Reportedly, fixes over Kernal 2 are:

1) The INPUT command has been fixed so that the INPUT prompt is not included with the response when the prompt is greater than 40 characters.

2) The problem with DELeting the last character of the last line on the screen has been corrected. Recall, if you start typing on the last line of the screen for 80 characters such that the screen scrolls twice, and then use DEL to move back and delete the 80th character, the CIA that lies above the colour table is disturbed and becomes very unfriendly. Now eliminated.

3) A problem was found in the RS–232 routines that occurred with either even or odd parity enabled that could result in inaccurate status reads.

4) Serial Bus Timing has been slightly modified to allow for several chained peripherals. When too many peripherals were connected on the serial bus the system would occasionally misbehave.

To test for Kernal 3, PRINT PEEK(65408). Details of price and availability are not yet available – call your local dealer or Commodore Service.

Cylinder Screen

For an interesting but useless screen effect on your 8000/9000 series machine, try this POKE from Dave Gzik of Burlington, Ontario:

```
POKE 59521, 40
```

When the video chip recovers from this punch you'll notice that your screen has been twisted into a cylinder. Reset or PRINT CHR$(14) will restore order.

Down Scroll 64 [155, 222]

Another of Murphy's unwritten laws states that "while trying to accomplish a specific task you will always accomplish some other task that brings you no closer to your original goal". Paul Blair of Holder, Australia has reconfirmed this law with the following submission.

". . .came across this while doing something else – all the best discoveries happen that way. The routine will scroll the Commodore 64 screen down starting from line D ie. from the top line with D = 0, second line with D = 1, etc. Colour changes from line to line are also allowed. At the end of the routine, some pointers are left a bit untidy, so use with caution. A PRINT or two on the end seems to restore order. . . thought you might like it – regards, Paul Blair".

```
100 d=0 : x=211 : v=15 : a=53280
110 poke a, 1 : poke a+1, 3
120 print" S ";
130 read a$ : v=v-1 : poke a, v : if a$=" end" then end
140 print" S ";
150 for t=1 to 10 : print a$;d : next
160 for t=0 to 14 : poke x+3, d : sys 59749 : next
170 print : for dl=1 to 2000 : next : d=d+1 : goto 130
180 data " scroll down with this pgm "
190 data " it's really very easy to use "
200 data " include it in games and so on "
210 data " list the pgm to see the set up "
220 data " see how you can select scroll start? "
230 data " have fun . . . . . paul blair "
240 data " end "
```

Equivalent VIC 20 and BASIC 2.0 routines have not been investigated but presumably would work depending on their ROMs. Fat 40 and 8000/9000 series machines don't need a routine like this – use PRINT CHR$(153) instead.

Screen Spaced With Colour Mods

Remember Screen Spaced? It has since been updated by Louis Black of Oshawa, Ontario to include colour on the C64. A VIC 20 version would not pose too big a problem. . . just swap out the numbers that reflect the screen width and address locations, as well as the POKEs in line 4 for border and background colours, and swap in the appropriate VIC 20 equivalents. Lines 1 and 3 must be entered using abbreviated keywords on at least some of the commands to make them fit on one line.

```
0 print" S ";
1 c=32: forn=1to41: gosub3: c=192-c: fora=0ton
  : forb=1024+ato2024stepn: pokeb,c: nextb,a,n
2 end
3 x=int(15*rnd(1)): y=int(15*rnd(1)): poke53280,x: poke53281,y
  : fori=1to100: next: return
```

Machine Language Screen Spaced

Here's Screen Spaced in machine language for the 64. Writer Chris Zamara said he had to insert a delay loop into the code because it was just too fast to have any adverse effect on your brain. Although it's still faster than the BASIC version, you will also notice that it's much smoother. Once again, the Surgeon General advises that danger to mental health increases geometrically with the number of SS iterations. And as they say on the 20-Minute Workout, "do not over Space yourself". And Murphys' first law says, "if something can go SS, it will". And Mr T. says "jus try it, fool"

```
1000 rem machine code screen spaced
1010 for j = 49152 to 49330 : read x
1020 poke j, x : ch = ch + x : next
1030 if ch <> 24671  then print " checksum error "  : end
1040 sys 49152 : goto 1040
1050 data  76,   9, 192,   0,   0,  32,   0,   9
1060 data  50, 160,   0,  32, 129, 192, 169,  32
1070 data 141,   5, 192, 169,   1, 141,   6, 192
1080 data  32,  52, 192,  32, 129, 192, 165, 197
1090 data 201,  63, 240,  10, 238,   6, 192, 173
1100 data   6, 192, 201,  42, 144, 234, 169,   0
1110 data 141,  33, 208,  96, 173,   5, 192,  73
1120 data 128, 141,   5, 192, 169,   0, 141,   3
1130 data 192,  32,  82, 192, 238,   3, 192, 173
1140 data   3, 192, 205,   6, 192, 240, 242, 144
1150 data 240,  96,  24, 173,   3, 192, 105,   0
1160 data 133, 253, 169,   0, 105,   4, 133, 254
1170 data 173,   5, 192, 145, 253,  32, 169, 192
1180 data  24, 165, 253, 109,   6, 192, 133, 253
1190 data 165, 254, 105,   0, 133, 254, 201,   7
1200 data 144, 230, 165, 253, 201, 192, 144, 224
1210 data  96, 169,   0, 133, 251, 169, 216, 133
1220 data 252, 173,   7, 192, 145, 251, 230, 251
1230 data 208,   2, 230, 252, 165, 252, 201, 219
1240 data 144, 239, 165, 251, 201, 232, 144, 233
1250 data 173,  33, 208, 205,   7, 192, 208,   0
1260 data  96, 174,   8, 192, 234, 234, 234, 202
1270 data 208, 250,  96
```

amaZAMARAing

(Sorry Chris – I just couldn't resist it) Here's another blitzoid screenzler: Timescroll for the C64 from Chris Zamara of Downsview, Ontario. Notice how the line is padded with spaces in two spots? Change the number of these

spaces for different effects. Line 20 details the exact number to start with. You can also change variable R to 53280 (the border colour register) for madded adness.

```
10 a=0:b=1:r=53281:fori=0to1step0:    poker,a:        poker,b:next
20 rem step0:3 spaces poker,a: 7 spaces poke etc.
```

Quick Note: The VIC 20, matched task for task, is the fastest of the Commodore machines. (circa June 1984)

Stop RUN/STOP [107, 143]

Most of you have no doubt seen at least one RUN/STOP disable for the C64. The following POKE was published several issues ago. It disables RUN/STOP (and RUN/STOP–RESTORE) without affecting the TI clock, but don't try LOADing or SAVing and expect normal results!

```
POKE 808, PEEK(808)–16
```

Therefore, this should only be used after the program has been LOADed and only with programs that do not LOAD subsequent software modules. This next routine is by James Whitewood of Milton, Ontario. It does everything the above POKE does without messing up LOAD and SAVE:

```
10 lo=12 * 4096
20 c=int(lo/256) : b=lo–c*256
30 for i=lo to i+4 : read a : poke i,a : next
40 poke 808, b : poke 809, c : end
50 data 169, 255, 133, 145, 96
```

The address computed in line 10 as variable LO can be any available memory ie. the cassette buffer will host this routine just fine. Notice how line 30 uses the loop variable I in the calculation "I+4" to specify the end of the loop. This is quite legal since I is set to LO and entered in the simple variables table (just like any other variable) before BASIC interprets the TO operative. However, you might also notice that 4 is one less than the number of DATA items. In situations like these, inclusive logic must be used to determine the number of loop iterations.

Cursed Commodore Cursor! [64]

Keith Preston of Ottawa, Ontario, has these comments on invoking the built–in cursor routines while a program is running, as detailed in The T.

"Several articles in Volume 4, Issue 6 suggest that a flashing cursor, the neophyte's comforter, may be retained during a Commodore GET by invoking POKE 204, 0. These are "Auto Liner" on page 18, "Subroutine Eliminators" on page 37 and "Three GET Subroutines" on page 38. When using the C64, however, the single POKE does not guarantee a flashing cursor for more than the first character of an input string (as requested in "Auto Liner"). Furthermore, the cursor may disappear upon hitting RETURN! To prevent this, simply add:

```
POKE 207, 0
```

in any line after the GET. A further:

```
POKE 204, 1 : POKE 207, 0
```

before exiting the input routine ensures a return to normal cursor function.

The accompanying short routine illustrates the technique and should be used to replace "Auto Liner". A number of other minor errors in that program have also been corrected."

```
60000 input "64 auto: start, increment";s,i
60010 print "■■■■■";s;:poke204,0
60020 geta$ : if a$ = "" then 60020
60030 poke 207, 0 : print a$; : if asc(a$)<>13 then 60020
60040 p = peek(1145 + len(str$(s))) : if p = 32 or p = 160 then 60010
60050 print "s = "s+i":i = "i":goto60010■"
60060 poke 631,13 : poke632,13 : poke198, 2
60070 poke 204, 1 : poke 207, 0 : end

60000 input "4.0/2.0 auto: start, increment";s,i
60010 print "■■■■■";s;:poke167,0
60020 get a$ : if a$ = "" then 60020
60030 poke 170, 0 : print a$; : if asc(a$)<>13 then 60020
60040 p = peek(33009 + len(str$(s))) : if p = 32 or p = 160 then 60010
60050 print "s = "s+i":i = "i":goto60010■"
60060 poke 623,13 : poke624,13 : poke 158, 2
60070 poke 167, 1 : poke 170, 0 : end
```

Sorry, But That DOES Compute

Ernest Blaschke of Sudbury, Ontario has these comments:

"In the commercial world, we all have heard the phrase: "Sorry, the computer made a mistake!". We know, of course, that it is the programmer and not the computer that made the mistake. Computers don't make mistakes. Right?

Well let me show you that your computer will make mistakes and will logically contradict itself. Yet, not all is lost. A programmer should know the computers' weaknesses and keep it from making true mistakes.

Type into your computer the direct command:

```
PRINT 5↑8
```

The reply will be 390625. The computer has in fact produced the correct value which is 5*5*5*5*5*5*5*5

Now enter the following small program:

```
10 if 5↑8  = 390625 then print "true"
20 if 5↑8 <> 390625 then print "false"
```

Type "RUN" and the computer will print "false", contradicting its previous statement that 5↑8=390625.

You probably know that your computer will reply with –1 to a true statement and with 0 to a false one.

If you aren't sure about this, try:

```
PRINT (2*2=4)
```

The computer replies with –1 (true). PRINT (2*2=5) will result in 0 (false). However, even using this approach, the computer stubbornly denies its own findings that 5↑8=390625.

PRINT (5↑8=390625) will reply with 0, false.

So what happened? The problem is that the computer calculates 5↑8 in floating point arithmetic and due to roundoff errors thinks the result is slightly greater than 390625. For printing, it "rounds off" the value in memory to the correct 390625. However, equality tests fail since the computer perceives the true result to be larger, and therefore unequal.

There are whole sets of problems where it is essential for the programmer to avoid this pitfall in order for the computer to do its task reliably. Bearing potential roundoff errors in mind, the programmer should have typed:

```
PRINT (INT(5↑8)=390625)
```

This would result with the –1 or true response. Of course this is limited to numbers that can be anticipated to have no fractional content. For numbers

with magnitude to the right of the decimal point, the programmer should consider moving the decimal point right by multiplying by some multiple of 10, say 100 or 1000, or as many significant digits as desired. Then take the INTeger portion of this number and divide by the same multiple of 10.

I hope to have convinced you that you may not blindly trust everything that appears on the screen or consider your computer's logic infalible."

Low-Res Screen Copy [149]

If you've ever attempted to do a low resolution screen dump of a screen containing graphics, you've seen that the printer leaves a little horrible space on carriage returns. This leaves the printout looking like it went through a shredder. But by using "LOW RES COPY", you can eliminate that space on the printout. The program itself is only 14 lines long, somewhat shorter than the 22 lines of "Screen Copy" in the VIC 1525 user's manual. I find this program to be a very handy utility when the time arises that you need a true low resolution screen copy. Brian Dobbs, Timmins, Ontario.

```
100 si$ = chr$(15) : bs$ = chr$(8) : d = 1024 : open4,4
110 for a = d to d + 39
120 print#4, si$;
130 b = peek(a)
140 if b>-1 and b<32 then e$ = chr$(b + 64)
150 if b>31 and b<64 then e$ = chr$(b)
160 if b>63 and b<96 then e$ = chr$(b + 32)
170 if b>95 and b<128 then e$ = chr$(b + 64)
180 print#4, e$;
190 next
200 print#4, bs$
210 d = d + 40 : if d>1984 then 230
220 goto 110
230 end
```

Eep Eep [81, 82, 111, 146, 208]

Eep Eep is a short interrupt driven routine that uses the cursor countdown timing register to drive the CB2 transducer (it's not really a speaker so it's called a transducer). Eep Eep only works on BASIC 4.0 machines but could be modified to drive the SID or VIC 20 sound registers. However, it's only good for two things really: one, it demonstrates the concept of pre-interrupt code. Notice the first 9 numbers in the DATA statements – you can almost read them without a dissassembler. They go LDA with 131, STA in location 144, LDA with 2, STA in location 145, and RTS (96). 2 times 256 plus 131 equals 643

which is where the actual pre–interrupt program begins (LDA with 16 right after the RTS). This is one of the most common methods to engage a pre–interrupt routine, and the quickest ways to spot one – something to remember when you find some old listing lying around.

At the end of line 1090 are three 234's. These are NOPs. It means simply No OPeration or NO oPeration, whichever you prefer. The reason these are here is to accommodate the three POKEs in line 1035. Line 1035 can be left out for a different Eep Eep. RUN the program as is, then remove 1035 and RUN again.

Line 1100 contains the code JMP to location $E455. This is the regular interrupt routine that the computer usually goes to when there is no pre–interrupt code – another way to spot pre–interrupt routines.

Eep Eep plays with the same chip responsible for LOADs and SAVEs. It's suggested you purge your machine of Eep Eep before continuing with more serious work.

Oh ya, the other thing Eep Eep does effectively is drive you bonkers. Just hook your computer up to your stereo, start Eep Eep, and tell no–one to touch your equipment. Then leave.

```
1000 rem eep eep – rte 1984
1010 for j=634 to 676 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 6145  then print " checksum error "  : end
1035 poke 671, 238 : poke 672, 147 : poke 673, 2
1040 sys 634
1050 data 169, 131, 133, 144, 169,    2, 133, 145
1060 data  96, 169,  16, 141,  75, 232, 169,  20
1070 data 141,  74, 232, 165, 168, 141,  72, 232
1080 data 160,    0, 200, 208, 253, 169,    0, 141
1090 data  75, 232, 141,  74, 232, 234, 234, 234
1100 data  76,  85, 228
```

Mirror

Mirror is another pre–interrupt routine also written by Richard Evers. It was written for no other reason but to see it work.

```
1000 rem mirror 40 – rte 1984
1010 for j=634 to 682 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 6710  then print " checksum error "  : end
1040 sys 634
```

```
1050 data 169, 131, 133, 144, 169,   2, 133, 145
1060 data  96, 162,   0, 160, 255, 189,   0, 128
1070 data 153, 232, 130, 136, 232, 208, 246, 238
1080 data 137,   2, 206, 140,   2, 173, 137,   2
1090 data 201, 130, 208, 233, 169, 128, 141, 137
1100 data   2, 169, 130, 141, 140,   2,  76,  85
1110 data 228
```

```
1000 rem mirror 80 - rte 1984
1010 for j=634 to 682 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 6696  then print "checksum error" : end
1040 sys 634
1050 data 169, 131, 133, 144, 169,   2, 133, 145
1060 data  96, 162,   0, 160, 255, 189,   0, 128
1070 data 153, 208, 134, 136, 232, 208, 246, 238
1080 data 137,   2, 206, 140,   2, 173, 137,   2
1090 data 201, 132, 208, 233, 169, 128, 141, 137
1100 data   2, 169, 134, 141, 140,   2,  76,  85
1110 data 228
```

The C64 version is a little longer due to colour table servicing required for Kernal 2 machines. However, it stops working after a Clear Screen is done, until the POKE in line 1040 is given. Can someone help us here? It's probably just some silly oversight that we can't seem to spot because of the clouds between us and the screen – you know the ones we mean, they're made of clear air? Hmm.

```
1000 rem mirror 64 - rte 1984
1010 for j=828 to 900 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch<> 8190  then print "checksum error" : end
1040 poke 53281, 493-peek(53281) : sys 828
1050 data 169,  71, 141,  20,   3, 169,   3, 141
1060 data  21,   3,  96, 162,   0, 160, 255, 189
1070 data   0,   4, 153, 232,   6, 189,   0, 184
1080 data 153, 232, 186, 136, 232, 208, 240, 238
1090 data  77,   3, 206,  80,   3, 238,  83,   3
1100 data 206,  86,   3, 173,  77,   3, 201,   6
1110 data 208, 221, 169,   4, 141,  77,   3, 169
1120 data   6, 141,  80,   3, 169, 184, 141,  83
1130 data   3, 169, 186, 141,  86,   3,  76,  49
1140 data 234
```

Ram Scan might be useful to somebody out there. Once engaged, it continually displays as many bytes of memory as will fit on the screen. Positioned over Zero Page, it will show the various timers, etc, in action. Same with the VIA and PIA registers up at $E800. To move the display use the cursor keys – cursor up/down moves it by one line of bytes, cursor left/right by one byte at a time. The STOP key puts you back in BASIC. Other than this, it too will give some pretty eye crossing patterns, something Richard seems to enjoy inflicting. Try moving the display around just below, and then above the first screen address.

```
1000 rem ram scan 80 – rte 1984
1010 for j=634 to 724 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 11974  then print "checksum error"  : end
1040 sys 634
1050 data 165, 151, 201, 255, 240,  43, 166, 152
1060 data 224,   0, 208,  10, 201,  17, 208,  16
1070 data 238, 178,   2,  76, 171,   2, 201,  17
1080 data 208,  16, 206, 178,   2,  76, 171,   2
1090 data 201,  29, 208,   6, 238, 177,   2,  76
1100 data 171,   2, 201,  29, 208,   3, 206, 177
1110 data   2, 160,   0, 174, 178,   2, 185,   0
1120 data 255, 153,   0, 128, 200, 208, 247, 238
1130 data 178,   2, 238, 181,   2, 173, 181,   2
1140 data 201, 136, 208, 234, 142, 178,   2, 169
1150 data 128, 141, 181,   2, 165, 155, 201, 239
1160 data 208, 166,  96
```

```
1000 rem ram scan 40
1010 for j=634 to 744 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 14739  then print "checksum error"  : end
1040 sys 634
1050 data 169, 147,  32, 210, 255, 165, 151, 201
1060 data 255, 240,  41, 166, 152, 208,  10, 201
1070 data  17, 208,  16, 238, 199,   2,  76, 174
1080 data   2, 201,  17, 208,  16, 206, 199,   2
1090 data  76, 174,   2, 201,  29, 208,   6, 238
1100 data 198,   2,  76, 174,   2, 201,  29, 208
1110 data   3, 206, 198,   2, 173, 198,   2, 133
1120 data 251, 173, 199,   2, 133, 252, 169,  19
1130 data  32, 210, 255,  32,  23, 215, 160,   0
1140 data 174, 199,   2, 185,   0, 255, 153,   5
1150 data 128, 200, 208, 247, 238, 199,   2, 238
1160 data 202,   2, 173, 202,   2, 201, 132, 208
```

```
1170 data 234, 142, 199,   2, 169, 128, 141, 202
1180 data   2,  32, 225, 255,  76, 127,   2

1000 rem ram scan 64
1010 for j = 828 to 916 : read x
1020 poke j, x : ch = ch + x : next
1030 if ch <> 10348  then print " checksum error "  : end
1040 poke 53281, 493-peek(53281) : sys 828
1050 data 165, 203, 201,  64, 240,  42, 174, 141
1060 data   2, 208,  10, 201,   7, 208,  16, 238
1070 data 115,   3, 184,  80,  27, 201,   7, 208
1080 data  16, 206, 115,   3, 184,  80,  17, 201
1090 data   2, 208,   6, 238, 114,   3, 184,  80
1100 data   7, 201,   2, 208,   3, 206, 114,   3
1110 data 160,   0, 174, 115,   3, 185,   0, 255
1120 data 153,   0,   4, 200, 208, 247, 238, 115
1130 data   3, 238, 118,   3, 173, 118,   3, 201
1140 data   8, 208, 234, 142, 115,   3, 169,   4
1150 data 141, 118,   3,  32, 225, 255, 184,  80
1160 data 167
```

Crystal

Crystal is just a short little program that draws a crystaline pattern on your screen. Aside from that, it demos how very little code it takes to get something happening – something like a game layout, a game intro, or an attract mode for a game you may have just finished and thought you didn't have room for an attract mode feature.

Crystal also demonstrates a technique that all programmers should be used to or else get used to – portability. Some programs aren't suited to be run on all machines, but those that could potentially be run on any machine should include for the user all necessary conversion information. It doesn't take long and it's a courtesy that adds an extra professional touch.

```
100 rem crystal
110 rem 8000/9000 series  : sw = 80
120 rem 4000 + c64        : sw = 40
130 rem vic 20            : sw = 22
140 rem 4.0 basic         : ss = 32768
150 rem c64               : ss = 1024 (default)
160 rem vic 20            : ss = 7680 (default)
170 rem sw = screen width : ss = screen start
180 print " SN ";: ss = 32768 : sw = 80 : rem * place your variables here
190 x = 1 : y = 1 : dx = 1 : dy = 1
```

```
200 poke ss+x+sw*y,81 : poke ss+x+sw*y,91
210 x=x+dx : if x=0 or x=sw-1 then dx=-dx
220 y=y+dy : if y=0 or y=24    then dy=-dy
230 s=peek(ss+x+sw*y) : if s=91 then dx=-dx : poke ss+x+sw*y,
86 : goto210
240 goto 200
```

Number Base Converter [80, 116, 137, 145, 163, 191, 207]

This next program works on BASIC 4.0 machines only because it uses some internal ROM routines of the built in Machine Language Monitor which the other machines don't have. Quite simply, it will convert numbers from one number base to another that are in hexadecimal, decimal, or binary.

There are two internal ROM routines used here: the first, SYS HD (where HD=55124), inputs a hexadecimal number from the keyboard and places its high order and low order components in locations 252 and 251. The program takes over from there and uses variable NO to build a decimal representation (line 12 or 13).

The second, SYS DH (where DH=55063), is the MLM routine for outputing a hexadecimal number whose high order and low order components are in locations 252 and 251 (line 15 or 19).

```
0 rem save"@0:hex/dec/bin conv",8:verify"0:hex/dec/bin conv",8
1 rem * richard evers - march 8th 1984 - 4.0 only *
10 input"[SHIFT]h[REV]ex>dec, hex>[SHIFT]b[REV]in, [SHIFT]d[REV]ec>hex,
   dec>[SHIFT]B[REV]in, bin>[SHIFT]H[REV]ex, bin>[SHIFT]D[REV]ec";q$
11 print"[CLR]";:hd=55124:dh=55063:if q$="B" then
   input"[DOWN]decimal ";no:goto16
12 if q$="b"then print"[DOWN]hex val ";:syshd
   :no=peek(251)+256*peek(252):goto16
13 if q$="h"then print"[DOWN]hex val ";:syshd
   :printpeek(251)+256*peek(252):goto10
14 if q$="H" or q$="D" then input"[DOWN]binary number";bn$
   :goto17
15 input"[DOWN]decimal ";a:b=int(a/256):c=a-256*b:poke251,c
   :poke252,b:sysdh:goto10
16 print:a=32768:for c=1 to 16:b=int(no/a):printb;:no=no-b*a
   :a=a/2:nextc:goto10
17 a=0:c=1:for b=len(bn$) to 1 step-1:a=a+val(mid$(bn$,b,1))*c
   :c=c*2:nextb
18 if q$="D" then print"decimal"a:goto10
19 print"$";:b=int(a/256):c=a-256*b:poke251,c:poke252,b
   :sysdh:goto10
```

The Un-Cursor

Still another pre-interrupt routine is this one called Un-Cursor. As the name might imply, Un-Cursor flashes everything on the screen except the space at the cursor position. At least that was the original intention – the real cursor seems to slip in an appearance every once in a while.

These pre-interrupt routines we've been bombarding you with may have no place in your utilities library, but they do serve one vital purpose. By giving you several examples we believe we accomplish two things – eliminating the fear and apprehension of messing with the fundamental operation of the machine is an important step towards becoming proficient with your computer. And second, when you come up with your own idea for a pre-interrupt program, we hope one of these examples will serve as a guide to completing your task.

```
1000 rem un-cursor 80
1010 for j=634 to 692 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 7656  then print "checksum error" : end
1040 sys 634
1050 data 169, 131, 133, 144, 169,   2, 133, 145
1060 data  96, 165, 170, 201,   1, 240,  41, 169
1070 data 128, 133,  88, 169,   0, 133,  87, 168
1080 data 177,  87,  73, 128, 145,  87, 200, 208
1090 data 247, 230,  88, 165,  88, 201, 136, 208
1100 data 239, 238, 134,   2, 173, 134,   2, 201
1110 data   2, 208,   5, 169,   0, 141, 134,   2
1120 data  76,  85, 228
```

```
1000 rem un-cursor 40
1010 for j=634 to 692 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 7652  then print "checksum error" : end
1040 sys 634
1050 data 169, 131, 133, 144, 169,   2, 133, 145
1060 data  96, 165, 170, 201,   1, 240,  41, 169
1070 data 128, 133,  88, 169,   0, 133,  87, 168
1080 data 177,  87,  73, 128, 145,  87, 200, 208
1090 data 247, 230,  88, 165,  88, 201, 132, 208
1100 data 239, 238, 134,   2, 173, 134,   2, 201
1110 data   2, 208,   5, 169,   0, 141, 134,   2
1120 data  76,  85, 228
```

```
1000 rem un-cursor 64
1010 for j=828 to 888 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 6949  then print" checksum error "  : end
1040 sys 828
1050 data 169,  71, 141,  20,   3, 169,   3, 141
1060 data  21,   3,  96, 165, 207, 201,   1, 240
1070 data  41, 169,   4, 133,  88, 169,   0, 133
1080 data  87, 168, 177,  87,  73, 128, 145,  87
1090 data 200, 208, 247, 230,  88, 165,  88, 201
1100 data   8, 208, 239, 238,  74,   3, 173,  74
1110 data   3, 201,   2, 208,   5, 169,   0, 141
1120 data  74,   3,  76,  49, 234
```

```
1000 rem un-cursor 20
1010 for j= 828 to 888 : read x
1020 poke j, x : ch=ch+x : next
1030 if ch <> 7141  then print" checksum error "  : end
1040 sys 828
1050 data 169,  71, 141,  20,   3, 169,   3, 141
1060 data  21,   3,  96, 165, 207, 201,   1, 240
1070 data  41, 169,  30, 133,  88, 169,   0, 133
1080 data  87, 168, 177,  87,  73, 128, 145,  87
1090 data 200, 208, 247, 230,  88, 165,  88, 201
1100 data  32, 208, 239, 238,  74,   3, 173,  74
1110 data   3, 201,   2, 208,   5, 169,   0, 141
1120 data  74,   3,  76, 191, 234
```

Volume 5, Issue 03

Line Doo Daa [112]

Our first screen blitz was submitted by Giovani Polese of Downsview, Ontario. The program shown is somewhat longer than it has to be – try PRINTA$ after running it once. We changed it to make it enterable from all keyboards (business keyboards don't have some of the graphic characters available). Try changing the '15' in line 30 to 14, 13, 12, etc., for different effects.

```
30 for j = 1 to 15 : read a
40 a$ = a$ + chr$(a) : next
50 print" SN ";
60 print a$; : goto 60
70 data 164, 210, 198, 192, 195, 196, 197, 163
80 data 197, 196, 195, 192, 198, 210, 164
```

Colourtest [28, 33, 111]

Colourtest is a simple little program that merely draws boxes in all the colours available so that you can adjust your TV/monitor for the best possible contrast. Like the program above, it too is much longer than it needs to be. For example, lines 115, 120 and 125 can be replaced by C$ = all the colour control characters except black, which is the background colour used for the test. This will also save you from entering the DATA statements. Lines 135 and 140 can be replaced by B$ = 1 cursor down and 5 cursor lefts.

```
100 print" S " : rem clear screen
110 poke 13*4096 + 33,0
115 for i = 1 to 15
120 read a : c$ = c$ + chr$(a)
125 next
130 s$ = "     ":rem 5 spaces
135 b$ = chr$(157)
140 b$ = " a " + b$ + b$ + b$ + b$ + b$
145 for i = 1 to 15
150 print mid$(c$,i,1);
155 print " s " ; : rem home
160 for j = 1 to i:print:next
170 printspc(i*2) " r " s$b$s$b$s$b$s$
175 for j = 1 to 300:next
180 next i
200 data   5,  28, 159, 156,  30,  31
210 data 158, 129, 149, 150, 151, 152
220 data 153, 154, 155
```

Would You Buy A Used Car From This Man?

Bytefinder

Have you ever needed to know what byte values are NOT present in a program or file you may be working on? The situation arises when you need a value to act as a terminator. If this same value exists elsewhere, the file will be terminated prematurely. The following program will show which values are not present in the 4K ROM block between $F000 and $FFFF. Quite simply, the program counts the occurence of byte values (line 120) by incrementing the approriate array element of U(. Naturally, all the values will lie between 0 and 255, hence DIM U(255). The elements of U(that remain zero indicate values that were not encountered (line 210).

```
100 dim u(255)
110 for j=15*4096 to 65535
120 x=peek(j):u(x)=u(x)+1
130 next j
200 for j=0 to 255
210 if u(j)=0 then print j;
220 next j
```

This could be easily altered for any area of memory, or for any disk file by changing:

```
110 open 8,8,8, "some file"
120 get#8, a$ : sx=st
125 x=asc(a$+chr$(0)) : u(x)=u(x)+1
130 if sx=0 then next j
140 close 8
```

Quick Note: Remember, a COLLECT D0 or OPEN 1,8,15,"V0" never
[213] **hurts, especially after you see something strange happen. You know what to change for drive 1.**

UN–DIMension

As you know, any attempt to DIMension an array that is already in use will result in the REDIM'D ARRAY ERROR. In fact, the only way to DIM an array by the same name twice is to issue a CLR which destroys all your other variables.

In most cases you shouldn't have to define an array more than once. But sometimes a program may lack memory for a particular operation because some array that isn't required is occupying valuable space. The program would be required to determine if array definitions could be erased without losing valuable information. Then, using the following techniques, some or all of the arrays could be eliminated. After performing the sort, etc., the arrays can be re–defined, ready for further use.

In another case, you may have an array that is too small. When your program detects this, invoke UN–DIM and re–DIM the array (by the same variable name) at the new larger size.

This method can not quite be called 'dynamic dimensioning'. First, you must actually eliminate the array before it can be re–defined. Any important data contained in the target array must be re–established after it is re–DIMed. Secondly, you cannot eliminate an array without affecting other arrays defined at a later time. In other words, the last array defined will be the first one erased, and so on. Therefore, it is best to DIM the arrays first that will be considered permanent and DIM the "variable" arrays last.

Function A(Q) (line 100) measures the "distance" in bytes from the Start of Arrays Pointer to the End of Arrays Pointer. When new simple variables are defined, both these pointers change as the arrays get pushed higher in memory. But the size of the arrays hasn't changed. So to erase an array, you simply back up the End of Arrays Pointer by the same distance (line 140). BASIC only looks up to the End Pointer for existing arrays, so if it isn't found DIM is allowed.

The next program is an "untaxed" and less commented version of the program after it.

VIC 20 / Commodore 64 Version (For BASIC 2.0/4.0 subtract 3 from all PEEK/POKE address in the first 5 lines.)

```
100 def fn a(q) = (peek(50)-peek(48))*256 + peek(49)-peek(47)
110 def fn hi(q) = peek(48) + int(q/256)
120 def fn lo(q) = peek(47) + (q and 255)
130 goto 160
140 poke 50, fn hi(x) : poke 49, fn lo(x) : return
150 rem * start of program *
160 dim a(10), c(15), b(15) : a(3) = fna(0)
170 dim j(20), i(20)        : a(5) = fna(0)
180 x = a(3) : gosub 140    : rem clr j( & i(
190 dim j(100),i(100)       : a(5) = fna(0) : rem re-dim
200 dim ad(250)
210 x = a(5) : gosub 140    : rem clr array ad(
220 x = a(3) : gosub 140    : rem clr j( & i( arrays
230 x = 0    : gosub 140    : rem clr all arrays
```

BASIC 2.0/4.0 Version (For VIC/64, add 3 to all PEEK/POKE addresses in first 6 lines.)

```
100 def fn a(q) = (peek(47)-peek(45))*256 + peek(46)-peek(44)
110 def fn hi(q) = peek(45) + int(q/256)
120 def fn lo(q) = peek(44) + (q and 255)
130 goto 180
140 rem --- clr array subroutine ---
150 poke 47, fn hi(x) : poke 46, fn lo(x)
160 return
170 rem *** start of program ***
180 dim a(10), b(15), c(15) : a(3) = fna(0)
190 rem a(3) = bytes used by first 3 arrays, a(, b( & c(
200 p = 3.14159 : i% = 10 : etc$ = " and so on "
210 rem arrays move up as simple variables are defined
220 rem however, a(3) remains the same
230 dim j(20),i(20)          : a(5) = fna(0)
240 rem new arrays, a(5) = bytes used by all 5
250 r$ = chr$(13) : q$ = chr$(34)
260 rem and perhaps some new variables
270 x = a(3) : gosub 150
280 rem clr arrays j( & i(, leaving a(, b( & c( intact
290 dim j(100),i(100)        : a(5) = fna(0)
300 rem re dim j( & i(
310 dim ad(250)
320 x = a(5) : gosub 150 : rem clr last array
330 x = a(3) : gosub 150 : rem clr j( & i( arrays
340 x = 0    : gosub 150 : rem clr all arrays
```

ERROROUTER [82, 133, 198] Scott MacLean, Toronto

Many people have written programs that they do not want to have other people crash out of either by accident or on purpose. The short program presented here traps all errors and re-runs the program if an error occurs. The program is written in BASIC, with a machine language routine loaded with data statements. It will work on the VIC or 64. Run the program and it will ask; "Install where?". Enter an address of safe RAM in your computer (see below). When you press RETURN it will enter the machine language section and activate it.

Safe places to install

C64	49152 or 828
VIC(5K)	7168 or 828
VIC(+8K)	16354 or 828
VIC(+3K)	7168 or 828

Location 828 is the tape buffer. Use it only if you are not doing any tape operations, otherwise the computer will crash when you get an error. To use this routine in your own programs, enter the data statements and read them into free RAM. Then poke locations 768 and 769 with the LO/HI address of the place you put the program in. It will then be activated.

How it works

Locations 768 and 769 are the locations which tell the computer where to go if it encounters any kind of error. By POKEing these locations with our own numbers, we can tell the computer to execute our own program instead of it's regular error routine. This program POKEs the numbers representing RUN and a chr$(13) (return) into the keyboard buffer. Then it jumps to the normal error routine. The computer then displays the error and checks the buffer. It sees some characters there and assumes the user typed them, so it displays and executes them, thereby re-RUNning the program in memory.

This program could be used for just about any program you write, it makes it virtually crashproof. I use it on my bulletin board, so if someone manages to crash it, it simply restarts itself, hanging up on the user in the process. I'm sure you'll find many other uses.

```
5 l = peek(768) : h = peek(769)
10 data 169, 82, 141, 119, 2
15 data 169, 85, 141, 120, 2
20 data 169, 78, 141, 121, 2
25 data 169, 13, 141, 122, 2
30 data 169,  4, 133, 198, 76, 256
35 print chr$(147);
```

```
40 input " install where " ;x : y = x
50 read a
55 if a = 256 then 75
60 ck = ck + a
70 poke x,a : x = x + 1 : goto50
75 poke x, l : poke x + 1, h
80 if ck<>2568 then print " data error " : end
90 hi = int(y/256) : lo = y-(hi*256)
100 print " installed at " y
110 poke 768, lo : poke 769, hi : new
```

Line Hider

Line Hider does just that – hide lines of code that you don't want shown without affecting their operation in the program. However, if you use Line Hider to hide a line that is the target of a GOTO or GOSUB, you'll get an UNDEF'D STATEMENT ERROR. Use the next utility for these.

There's just one trick to using it – you must supply the input with the number of the line that comes BEFORE the one you wish to hide. It wouldn't be hard to modify this to hide an entire program!

```
100 rem  save " @0:line hider " ,8:verify " 0:line hider " ,8
105 rem  * hide a line within your basic program
110 rem  * basic 4.0 : sb = 1025
115 rem  * c64 only  : sb = 2049 (default)
120 rem  * vic only  : sb = 4097 (default)
125 :
63989 sb = 1025 : rem ** set-up for basic 4.0
63990 input " line # of preceding line " ;pl
63991 for lp = 1 to (2↑16)-1
63992 num = peek(sb + 2) + peek(sb + 3)*256 : rem * line #
63993 nxt = peek(sb) + peek(sb + 1)*256
63994 if num < pl then sb = nxt : next lp : end : rem * still below the line
63995 if num > pl then print " line not found " : end
63996 sh  =  peek(sb) + peek(sb + 1)*256  : rem * position of line to hide
63997 nl = peek(sh) : nh = peek(sh + 1) :rem ptrs to next line
63998 poke sb,nl : poke sb + 1,nh :rem bypass the line to hide
63999 poke sh + 2,0 : poke sh + 3,0 :rem and change line # to zero
```

Ghost Liner

Ghost Liner does just what Line Hider does, except the line number will be displayed with nothing beyond it. Ghost Liner searches for lines that start with 5 colons. It substitutes the first colon with a zero. When the LIST routine sees

this zero, it assumes end of line and goes on to list the next line. RUN is not affected.

```
100 remark * ghost liner - rte
110 remark * cloaks all lines starting
120 remark * with ::::: (5 colons)
130 remark * basic 4.0 : vl = 42 : vh = 43 : sb = 1025
140 remark * c64 & vic : vl = 45 : vh = 46
150 remark * c64 only  : sb = 2049 (default)
160 remark * vic only  : sb = 4097 (default)
170 :
180 : vl = 42:vh = 43:sb = 1025 : rem * basic 4.0 set-up
190 loc = peek(vl) + 256*peek(vh)
200 printchr$(147)loc, " : maximum "
210 print, " : current "
220 if peek(sb)<>58 then 250
230 ct = sb : for lp = 0 to 0 : ct = ct + 1 : lp = (peek(ct) = 58) : next
240 if ct > sb + 4 then poke sb, 0 : sb = sb + 4
250 sb = sb + 1 : print chr$(19)chr$(17)sb : if sb<loc then 220
260 end
```

List Decorator

With all the screen function characters available for changing colour and cursor position, why not make use of them while LISTing as well as when you RUN. List Decorator will take dull, unoticeable remarks and make them bright and easy to spot. The list below shows what value to use for the possibilities. You need not stop at one though - after running it once on itself (see line 160 & 170), LIST the program and insert new @ signs in the same place. Now RUN again. List Decorator will replace all occurences of "REM @" with RB.

```
rb = 5   for white line (c64 & vic)
rb = 7   for ring the bell (cbm only)
rb = 13 for carriage return
rb = 14 for upper/lower case
rb = 15 to set the top left corner (cbm only)
rb = 17 for cursor down
rb = 18 for reversed program rem lines
rb = 19 for cursor home
rb = 20 for delete char
rb = 21 for delete a line (cbm only)
rb = 25 for scroll down (cbm only)
rb = 28 for red line (c64 & vic)
rb = 29 for cursor right
rb = 30 for green line (c64 & vic)
rb = 31 for blue line (c64 & vic)
```

```
100 rem * list decorator - rte
110 rem * lb = 42 : hb = 43 : sb = 1025 :rem * for basic 4.0
120 rem * lb = 45 : hb = 46      :rem * for c64 & vic
130 rem * sb = 2049              :rem * for c64 (default)
140 rem * sb = 4096              :rem * for vic (default)
150 :
160 rem @ this is how your remark should look when entered
170 rem @ every occurence is substituted
180 :
63995 lb=42 : hb=43 : sb=1025 : rem basic 4.0 setup
63996 input" replacement byte for @ ";rb
63997 mx=peek(lb)+peek(hb)*256 : for a=sb to mx
      : b=peek(a) : if b<>143 then285
63998 if peek(a+1)=32 and peek(a+2)=64 then poke(a+2),rb
63999 next : end
```

Sinhibitors [68, 179]

This next collection of handy POKEs was submitted by Adam Foster of Kingston, Ontario.

Many software companies go through a great deal of trouble to stop program pirates from stealing their software. But no matter how much protection you have on a program, if the pirate really wants to get in, he will.

On the VIC 20 and Commodore 64 there are several easy POKEs to stop the common thief. I stress the word "common" since any experienced pirate will get by these easily.

List Terminator

This feature will prevent others from viewing your program. On both the VIC and the 64 add a line to:

POKE 775, 200

To re-enable LIST, POKE 775 with 167 on the 64 and 199 on the VIC. Unfortunately, it only works if the program has been RUN before they try and LIST it.

Save Terminator [143]

The 64 version of this stops the saving of your program by disabling the RUN STOP/RESTORE keys. To do this:

POKE 808, 225 : POKE 818, 32

To return to normal POKE both locations to 237. On the VIC, this killer is enabled by:

```
POKE 802, 0 : POKE 803, 0 : POKE 818, 165
```

and is disabled with:

```
POKE 802, 243 : POKE 803, 0 : POKE 818, 133
```

STOP Key [88, 143]

To disable the STOP key, add:

```
POKE 808, 225
```

to your program. POKE 808, 237 turns the STOP key on again. This works on both computers.

Keyboard Killer

```
POKE 649, 0
```

turns the keyboard off, and POKE 649, 10 turns it back on for both VIC and 64.

Etch. . .,., A Sketch.

Not the quickest hi-res graphic aid, but it demonstrates clearly some fundamentals. Like setting up the hi-res screen, testing boundaries and adjusting for max/min, calculating hi-res position to the bit, testing for the fire button, and determining joystick direction. It wouldn't be tough to make this machine language. Written by Dave Gzik, Commodore Canada.

ETCHASKETCH [173]

Here is a neat little program that converts your C64 into an etcha-sketch type tablet. To use this, just load the program and run it. You'll need to have a joystick plugged into port 2.

Drawing is accomplished by moving the joystick in the direction you want and this program will draw in eight directions. If you want to lift the drawing pen just hold down the FIRE button and move where you want to go.

This is a very simple BASIC program, there is no cursor to indicate the location of the pen, so you'll be guessing when you lift it off the drawing area.

You can expand on this if you wish but it is rather slow in BASIC. Give it a try it's not that long or tedious.

```
5 rem etchasketch by dave gzik
10 base = 2*4096 : poke 53272, peek(53272)or8
20 poke 53265, peek(53265) or 32
30 for i = base to base + 7999 : poke i, 0 : next
40 for i = 1024 to 2023 : poke i, 3 : next
50 x = 160 : y = 100 : rem start off point
75 if y<0    then y = 199
76 if y>199 then y = 0
77 if x<0    then x = 319
78 if x>319 then x = 0
80 row = int(y/8) : char = int(x/8) : line = y and 7
90 bit = 7-(xand7): byte = base + row*320 + char*8 + line
95 if fr + jv = 111 then 110
100 poke byte, peek(byte) or 2↑bit
110 jv = 15-(peek(56320) and 15)
111 fr = peek(56320)
120 if jv = 1 then y = y-1 : goto75
140 if jv = 2 then y = y + 1 : goto75
150 if jv = 4 then x = x-1 : goto75
160 if jv = 5 then x = x-1 : y = y-1:goto75
170 if jv = 6 then x = x-1 : y = y + 1:goto75
180 if jv = 8 then x = x + 1 : goto75
190 if jv = 9 then x = x + 1 : y = y-1:goto75
200 if jv = 10 then x = x + 1 : y = y + 1 : goto 75
210 goto 75
```

Editor's Note: Notice how Dave tests the fire button in line 95. This works no matter what direction the joystick is being held. Why? Because the joystick ports are inverted logic. This means when nothing is happening on the joystick (except for the fact that it's plugged in) the joystick register will contain a value of 127 (bits 0–6 on, 7 off which flags port 2). Line 110 un–inverts the value by first looking at only the first 4 bits, and subtracting that from 15 to get direction values that make a lot more sense. As JV goes up FR goes down, so FR+JV remains constant, whether the fire is down or not. But when the fire button lS down, that constant is 111.

C64 Default Screen Colours [128, 144]

R.D. Young,
James Park, New Brunswick.

If your black and white TV has the blues, or at least if it doesn't like the blue default screen colours that appear on power–up, you can easily POKE in new colours. Then frequently and just as easily, you can watch your new colours disappear with each RUN–STOP/RESTORE key sequence and you must set them all over again. You may even have a favourite colour combination with your colour monitor. . . same problem.

Try the following little program. It loads a machine language program into any desired memory area, changes the "BASIC Warm Start Vector" to point there, and will keep your screen set to your own default colour combination.

The starting location for the machine language program is first selected. My default is decimal 900, the middle of the cassette buffer. Another usually safe place is between 49152 and 53232.

```
10 rem set default colours on run-stop/restore
20 rem by r.d. young
30 input "start location 900[left 5]";ad
40 for i=ad to ad+15 : read x : poke i, x : next
50 hi=int(ad/256) : lo=ad and 255
60 input "screen colour (0-15)  6  [left 3]";c
70 poke ad+1, c
80 input "cursor colour (0-15)  13 [left 4]";c
90 poke ad+9, c
100 poke 770, lo : poke 771, hi
110 sys 65126
500 data 169,   6, 141,  32, 208, 141
510 data  33, 208, 169,  13, 141, 134
520 data   2,  76, 131, 164
```

The defaults in the program are set to blue screen with light green text. Refer to any colour table (pg 159 in 64 User Guide) for colour codes that represent each colour choice. Both the screen and border are set to the same colour (my choice) but a little extra machine language could change all that. Happy RESTOREing!

Tape Saving Notes [126, 144, 164]

Saving to tape from BASIC merely writes to tape everything that lies between the Start and End of BASIC Pointers. Saving to tape from the Machine Language Monitor allows one to save any area of memory because the user supplies the start and end address. The format is:

```
sys 4   ;enter monitor on BASIC 4.0 machines

.s "some name",01,6000,7000
```

. . .which saves all memory from hex 6000 to 7000 on cassette #1 using the name "some name". But the MLM Save always had one drawback. It would not save any memory above hex 7FFF. The problem lies in the tape write routines that Commodore designed years ago with the PET 2001. Commodore assumed back then that tape would never be written with data above 7FFF. So

they used the high bit of the high byte of the address to signal end of write. When the current write address matched the end address (ie. End of BASIC Pointer), this bit would be set. The last byte would be output and, in a later part of the tape output routines, this bit would be detected and writing tape would be terminated. However, if the current write address goes above 7FFF, this bit is set naturally, but of course the tape close routine would have no way of differentiating and tape write would terminate.

Without telling anybody, it seems Commodore has lifted that restriction from the tape routines in the VIC 20 and Commodore 64. Although you must install your own MLM program (ie. Supermon, VICMON Cartridge, etc.) the following command will behave perfectly:

```
.s "some name",01,c000,d000
```

. . .will save to tape everything from $C000 to $CFFF. Remember, you must specify the last address desired, plus 1.

RESTORE X [48, 72, 223]

This short machine language loader was submitted by Garry Kiziak of Burlington, Ontario. It allows you to RESTORE the DATA pointer to any DATA line as opposed to the first DATA line. And with just one single SYS. Written for the 64 or VIC 20.

```
10 restr=828:for k=restr to restr+31:read j:poke k,j:next k
20 data 32,253,174,32,158,173,32,247,183,32,19,166,176,5,162,17
30 data 76,55,164,165,95,233,1,133,65,165,96,233,0,133,66,96
100 for i=1 to 20
110 x=100*(int(rnd(1)*5)+2)
120 sys restr,x
130 read a$ : print a$
140 next
150 end
200 data i'm at line 200
300 data i'm at line 300
400 data i'm at line 400
500 data i'm at line 500
600 data i'm at line 600
```

And yes, that was Jim. B., '69

Volume 5, Issue 04

64 Quick Beep [67, 82, 91, 146, 208]

The 64 is highly capable when it comes to sound generation, but it lacks a simple method of making a single beep, or ringing a "bell", as in the 40/8032 machines. The following POKEs will create a pleasant "ding", and can be used to get your attention after the computer has completed a certain task.

```
poke 54273,70: poke 54278,249: poke 54296,15
: poke 54276,17: poke 54276,16
```

Note: changing the argument in the first POKE varies the frequency of the ring.

Colour Bar [28, 33, 99]

(Credit for this goes to someone out there with a stylish but somewhat unreadable signature. . . M.S. Renouf, perhaps?)

"Recently while developing a colour select routine for a program of mine (colour of background, border, sprites, etc.) I discovered that certain colours side by side were virtually impossible to see. So I took a look at the Reference Manual, and using some sophisticated analysis methods (trial and error) came up with the best possible general colour map using all 16 colours strung out in a line side by side. If you POKE the colours below in consecutive screen positions, you will get a most readable Colour Bar:

```
10 data 4,0,8,2,10,9,7,12,6,3,14,1,11,13,5,15
20 c=55295 : s=1023
30 for j=1 to 16 : read a
40 poke s+j,160 : poke c+j, a : next
```

Dazzler of the Month

You were waiting for it, weren't you? Just so we don't disappoint you, here's a screen dazzler for any BASIC 4.0 machine (4032/8032):

```
10 for i=47 to 57 : poke 59521,i : for d=1 to 100 : next d,i : goto 10
```

Enter the above program and RUN it without clearing the screen. I call it "Attack of the Killer Program (in 3–D)". P.S. It doesn't look like it's very healthy for the video circuitry, so don't keep it running too long.

Which Way Did He Go?

Here's an effect that owes more to the nature of human visual perception than it does to the graphics capabilities of your computer. On any 40 column machine, enter this program:

```
10 print" ********************                    ";:goto 10
```

(Note that there are 20 asterisks and 21 spaces. The exact number of asterisks is not important, but there must be 41 characters altogether. You may use your favourite graphics symbol in place of the asterisks).

Now run it. You probably first see lines of asterisks running from the bottom of the screen to the top. Try fixing your eyes onto the centre of one of the bars of asterisks. See them moving slowly from left to right? You'll probably find the illusion flipping between vertically and horizontally moving bars.

The illusion is even more pronounced on 80–column machines. Use this program:

```
10 print" ****************************************";
20 print"                                        ";:goto10
30 rem 40 asterisks, 41 spaces
```

The 80–column version creates slow–moving bars that are very difficult to see as moving vertically. A procession of diagonal bars is seen moving slowly from left to right.

Aquarius [99]

While we're doing special effects, here's another dazzler for 80–column machines. It's based on the program by Giovani Polese in last issue's Bits & Pieces section, but works especially well on 80–column machines. In upper/lowercase text mode, enter:

```
10 print chr$(142)
20 print" EDCFRFCDE ";:goto20
```

I won't describe the resulting effect, but it's better than you'd expect from such a small program. TRY IT!

Quick Note: The more sprites you have displayed on the screen of the 64, the slower the processor operates due to wait states from the VIC chip. [197]

SHIFTing your WAIT [13, 127, 128, 134]

Here's a handy technique that comes to us from Rico Mariani of Downsview, Ont. To insert a pause in a program that can be enabled from the keyboard, use the command:

```
C64:      WAIT 654,1
40/8032:  WAIT 152,1
```

. . .which will wait until the shift key is pressed. For the opposite effect, you can wait until the shift key is released with:

```
C64:      WAIT 654,1,1
40/8032:  WAIT 152,1,1
```

To enable a program halt at that point, engage the shift/lock key. This is a good way to synchronize a program with an external process: just disengage the shift lock key to continue program execution. This way you can be certain whether or not the program will halt at the WAIT statement, simply by knowing the position of the shift lock key.

Interrupt Key-Scanning [158]

Sometimes it is desirable for some action to be performed any time a certain key is pressed. A routine may be set up to run during the interrupts, but the desired routine must be performed **once** when the key is depressed, not every interrupt as long as the key is held down. The following examples in assembler show an easy way to accomplish this.

C64 Example [128, 147]

The following assembler program, once initialized with SYS 49152, will change the border colour whenever the F1 key is pressed.

```
 10 *        =   $c000    ;start at 49152 decimal
 20 keybd    =   197      ;key pressed
 30 ;set up irq vector
 40          sei
 50          lda  #<intrtn
 60          sta  $0314
 70          lda  #>intrtn
 80          sta  $0315
 90          cli
100          rts
110 ;
120 prevkey  .byte 0
```

```
130 ;
140 intrtn      =   *
150             lda  #4         ;keyboard code for f1 key
160             cmp keybd       ;f1 key pressed?
170             bne  out        ;no, exit to system irq
180             cmp prevkey     ;check previous key pressed
190             beq  out        ;exit if f1 pressed previously
200 ;
210             inc  $d020      ;increment border colour register
220 ;(any desired code could be inserted here)
255 ;
230 out         =   *
240             lda  keybd
250             sta  prevkey
260             jmp  $ea31      ;system irq routine
```

40/8032 Example [128]

The example program for the CBM will switch between graphics/lowercase modes when the up-arrow key is pressed. Use the 64 program above, making the following changes:

```
10  *           =   $7000   ;start at 28672 decimal
20  keybd       =   151     ;key pressed
60              sta $90     ;irq vector low
80              sta $91     ;irq vector high
150             lda #222    ;keyboard code for up-arrow
210             lda $e84c   ;i/o register for graphics mode
212             eor #2      ;flip graphics mode bit
214             sta $e84c   ;store back into register
260             jmp $e455   ;system irq entry point
```

Use SYS 28672 to enable this version.

File Ripper [6, 173]

Need to look through some disk file in a real hurry? Actually File Ripper is far too fast for the eye, but if you want to see what's at the end of a large file and have no time to waste, File Ripper will get you there quick! Once at the point you're interested in, you can use the regular slowscroll or pause keys (back arrow, :, RVS, CTRL, etc.). The 64 version would theoretically work on the VIC 20 but it hasn't been tested.

```
1000 rem file ripper 4.0
1010 for j= 634 to 774 : read x
1020 poke j,x : ch = ch + x : next
```

```
1030 if ch<> 15942  then print " checksum error "  : end
1040 sys 634
1050 data 160,   2, 169, 209,  32,  29, 187,  32
1060 data 226, 180, 169,   0, 133, 218, 169,   2
1070 data 133, 219, 169,   5, 133, 210, 169,   8
1080 data 133, 212, 169,   5, 133, 211, 162,   0
1090 data 189,   0,   2, 240,   3, 232, 208, 248
1100 data 134, 209,  32,  99, 245, 166, 210,  32
1110 data 198, 255,  32, 207, 255,  32, 210, 255
1120 data 165, 150, 240, 246, 165, 210,  32, 226
1130 data 242,  32, 204, 255, 160,   2, 169, 238
1140 data  32,  29, 187,  32, 228, 255, 240, 251
1150 data 201,  89, 240, 172,  76, 255, 179, 147
1160 data  70,  73,  76,  69,  78,  65,  77,  69
1170 data  32,  63,  32,  40,  32,  68,  35,  58
1180 data  70,  73,  76,  69,  78,  65,  77,  69
1190 data  32,  41,  32,   0,  17,  18,  65,  71
1200 data  65,  73,  78,  32,  63,  32,  40,  32
1210 data  89,  32,  32,  79,  82,  32,  32,  78
1220 data  32,  41,  32, 146,   0
```

```
1000 rem file ripper 64
1010 for j= 828 to 926 : read x
1020 poke j,x : ch = ch + x : next
1030 if ch<> 11254  then print " checksum error "  : end
1040 sys 828
1050 data 160,   3, 169, 131,  32,  30, 171,  32
1060 data  96, 165, 169,   0, 133, 187, 169,   2
1070 data 133, 188, 169,   5, 133, 184, 169,   8
1080 data 133, 186, 169,   5, 133, 185, 162,   0
1090 data 189,   0,   2, 240,   3, 232, 208, 248
1100 data 134, 183,  32,  74, 243, 166, 184,  32
1110 data 198, 255,  32, 207, 255,  32, 210, 255
1120 data 165, 144, 240, 246, 165, 184,  32, 145
1130 data 242,  32, 204, 255,  76, 116, 164,  70
1140 data  73,  76,  69,  78,  65,  77,  69,  32
1150 data  63,  32,  40,  32,  68,  35,  58,  70
1160 data  73,  76,  69,  78,  65,  77,  69,  32
1170 data  41,  32,   0
```

Quick Note: to disable character set switching with the shift/Commodore keys, PRINT CHR$(8);. To re-enable, PRINT CHR$(9);. An easier way to do it is to simply key CTRL-H or CTRL-I, respectively. Thanks to Jeff Goebel for this latter tip.

[138, 175]

File Loader

When a number of program files must be loaded in succession, for example sprite or character definitions, machine code, or high resolution screens, this simple loading technique is a good way to do it:

```
10 A=A+1
20 ON A GOTO 30,40,50,60,70
30 LOAD"FIRST FILE ",8,1
40 LOAD"SECOND FILE",8,1
50 LOAD"THIRD FILE ",8,1
60 LOAD"FOURTH FILE",8,1
70 final statement – sys, load, goto, etc.
```

Since BASIC automatically performs a RUN (without clearing variables) after a LOAD from program mode, the files are loaded in succession. Any number of files may be similarly loaded, but make sure none of them are BASIC program files, or the loader program will get clobbered. As indicated, the last statement may be a SYS or other statement to start the program instead of a LOAD

ASCII/CBM Conversion [96, 137, 145, 163, 191, 207]

If you've ever tried to print to an ASCII printer, or receive from the RS–232 port on the 64, you're familiar with the problem: Upper and lower case are reversed. To solve the problem, use one of the following lines of BASIC to convert a single character, stored in A$.

ASCII to CBM

```
a=asc(a$+chr$(0)):a$=chr$(a+32*(a>96 and a<123)-128*(a>64 and a<91))
```

CBM to ASCII

```
a=asc(a$+chr$(0)):a$=chr$(a+128*(a>192 and a<220)-32*(a>64 and a<91))
```

Another difference between regular ASCII and CBM ASCII are the control characters. ASCII codes from 0 to 31 are reserved for special control characters, such as bell, linefeed, carriage return, backspace, etc. There is no direct correlation between ASCII and CBM control characters, but the conversion that must frequently be made is substituting the Commodore's "DELete" character (20) with ASCII's "backspace" (8). This may be done by adding the line,

```
if a=8 then a$=chr$(20)  For ASCII to CBM conversion, or
if a=20 then a$=chr$(8)  For CBM to ASCII conversion.
a$=chr$(a+12*((a=20)-(a=8)) ) will convert either way.
```

Any or all control characters may be converted from ASCII by setting up a conversion string which holds the desired CBM characters, such as cursor controls, tabs, etc. The position of each character in the string should correspond to the ASCI code that it replaces. To make the conversion using a conversion string 32 characters long, the following line could be added to the conversion program above:

```
if a<32 then a$ = mid$(c$,a + 1,1)
```

This technique may also be used to convert Commodore control characters into ASCII equivalents. Usually, however, only the delete/backspace characters need to be switched.

Quick Note: the GET statement can accept more than one argument, as in:

GET A$,B$,C$,D$, or using GET#

Easy Disk Salvaging [35, 181, 182]

All programmers live in constant fear of losing their irreplacable work due to death of a disk. This leads to paranoic backing up of important files, a very healthy activity. Occasionally, however, even the most paranoid among us hear the horrifying klik-klik-klik-klack-griiiind-klkklk which signifies – horror of horrors – a read error!

After you curse yourself for not having made a recent back-up, what can you do? Well, the first thing you should do before resorting to sector-reading, is to restore the disk jacket. A common cause of read errors is a disk jacket that's been squeezed tightly near the edges because of careless handling or storing. This creates too much friction between the disk and jacket, slowing the disk to the point where the drive can't read it. To fix this problem, carefully run the edge of the disk along the corner of a table to flatten it. Tapping the edge of the disk on a corner at many points also helps. This should spread out the jacket enough so that you can read the disk and make a copy of it onto a fresh one. That's the happy ending of this story, but there's also a moral: treat disks **gently** and don't hold them by the edges or squeeze them in any way. And that's not a fairy tale.

A Magic Number? [57, 63]

Examine the following program:

```
10 input " enter any number " ;n
20 print n
30 n$ = mid$(str$(n),2)
35 k = 0
40 for i = 1 to len(n$)
50 : k = k + val(mid$(n$,i,1))*3
60 next i
70 n = k : goto 20
```

As you can see, it just accepts any number, and then sums all of the digits in the number, first multiplying each digit by three. The result then becomes the new number, and the process repeats indefinitely, showing the new value, "N", every iteration. What's so special about it? Try it with any number you like, and see what happens after the first three or four iterations. If you can figure out the reason for this strange numerical omnipresence, your math students await you!

Safe VAL Function [7]

To permit "idiot proof" entry of numerical values from within programs, it is best to input a string, and then convert it to a floating point value with the VAL function. This technique is still not 100% idiot proof, however. If, for example, the string "1e99" is entered, attempting to take its VAL would result in an overflow error – disaster! The VAL of the string can only be taken if the result would not exceed 1.7e+38. The following subroutine will take the VAL of the string V$ and put the result in V if doing so would not cause an overflow error. If an error would result, the flag "OVERR" is set, and V is set to zero.

```
50000 rem* safe val subroutine
50001 rem* input parameter  : V$
50002 rem* output parameters: V, OVERR
50010 overr = 0: ll = len(v$)
50020 for ii = 1 to ll: if mid$(v$,ii,1)<>" e " then next ii
50025 if ii>ll goto 50065 :rem* no " e " s, ok
50030 : mn = val(mid$(v$,1,ii-1)) :rem* mantissa *
50040 : for jj = ii + 1 to ll: if mid$(v$,jj,1)<>" e " then next jj
50050 : ex = val(mid$(v$,ii + 1,jj-ii)) :rem* exponent *
50060 : if ex + log(mn) > 38.53 then overr = 1: v = 0: return :rem* too high
50065 rem— endif —
50070 v = val(v$) :rem* ok *
50080 return
```

Quick Note: An elegant way to create an infinite loop without using GOTO is:

FOR I = 0 TO 1 STEP 0 . . . NEXT I

Hardware Random Number Generation on the 64 [204]

From BASIC, it's easy to get random numbers (actually, pseudo–random numbers) using the RND function. If random numbers are desired in a machine language program, or if better randomness is desired, the SID chip may be used to supply them. The amplitude of the output waveform from voice three may be read from SID register 27, and if voice three is set up for high frequency noise generation, this value will be random. If that doesn't make sense to you, don't worry. Just set up the SID chip with:

POKE 54287,255: POKE 54290,129

Any time after that, a random number from zero to 255 may be read with:

PEEK(54299) from BASIC, or
LDA $D41B in assembler.

Round–up [62]

Here's some fun with floating point round–off errors that works with either BASIC 2.0 or 4.0. Enter:

?5.99999999 (8 nines)

The result, as you'd expect, is just what you entered. Now try:

?5.999999999 (9 nines)

This time the result is 6, which is quite reasonable, since it is just rounded off. But now go one step further and enter:

?5.9999999999 (10 nines)

What? Not so reasonable this time (try it). Before you trash your computer for being so stupid, don't worry: the floating point routines are accurate to about 7 decimal places – that's one part in 10 million!

Quick Note: I% = I is a quick way of taking the integer part of a variable without using the INT function.

Prime Number Generation

I know, I know, generating prime numbers probably isn't high on your list of fun things to do with a computer. Notwithstanding, you'll probably get a kick out of the following method and accompanying program. You may learn something, too – don't forget, this *is* the education issue.

Math class flashback: a prime number is a number which can only be evenly divided by 1 and itself. Thus, 11 is a prime because it has no factors other than 1 and 11, but 9 is not, since it's factors are 1, 3, and 9.

If you were asked to write a subroutine to determine whether or not a number is a prime, a reasonable approach would be to divide by each whole number from 2 up to the argument, and if none are found to divide evenly. . .

```
a = d/q: if a<>int(a) then. . .
```

. . .then the number is a prime. You could go one step further for efficiency and only try numbers up to the square root of the argument, since factors above that would be redundant. Now, here's another problem: write a program which prints all prime numbers from 1 up to a given value. You might be tempted to pass integers from one up to the limit to the above subroutine, and print the number if it is a prime. That will of course work, but there's a better, not–so–obvious way.

The prime number generation technique used here comes to us courtesy of Eratosthenes (E–RA–TOS'–THENEEZ), of Athens, Greece. Around 200 B.C., Eratosthenes had this great brainstorm for generating primes. The technology of the time did not include computers, so a long line of small stones was probably used to do the trick. Actually, it's no trick – here's how it works (We'll use a computer instead of the stones. Less work).

1) First we set up an array containing all zeros. The array must have as many elements as the maximum prime we want to generate. This array could be represented by a string of bits since only 0 or 1 is needed in any element.
2) We initialize the process by printing the first prime, which is 1, and setting the first element in the array accordingly.
3) The array is scanned from the current prime until a zero is found. The position of the next zero in the array represents the next prime, and it may be printed out. It will be 2 in this case.
4) The array element pointed to by this prime is set to a 1, and, in this first case, every second element thereafter is also set. In the next iteration (prime = 3), the third element and every third thereafter would be set.
5) Steps 3 and 4 are repeated until the next prime is greater than the maximum prime to be found.

The technique may look strange on initial examination, but if you think about it a bit, you'll see why it works. By setting a given element, you're ruling it's position out as a prime, and thus the multiples of every prime are being ruled out as primes. This effectively cancels out all numbers which have factors (other than 1 and the number itself).

Why go through mental gymnastics to generate primes? Well, this technique spits out primes so fast, you won't be able to read them as they fly by on the screen. The first few primes come out slowly, then get faster and faster as they approach the specified limit. The program below prints all the primes up to 100 (there are 26 of them) in about 4 seconds. An optimized BASIC program does it in less than three, much of that time taken just to print the numbers out.

Try different numbers for the maximum prime, and see the effects. Each array element is an integer variable instead of a single bit. A bit-oriented routine would allow higher primes to be generated since less memory would be required per prime, but would run slower due to increased processing.

```
100 rem* prime number generation
110 rem* using " sieve of Eratosthenes "
120 :
130 input " maximum prime " ;max
150 :
160 ti$ = " 000000 "
165 dim sieve%(max + 1)
170 number = 0: rem* prime count
180 prime = 1 : rem* first prime is 1
190 sieve(prime) = 1
200 :
210 for mloop = 0 to 1
220 : print prime
230 :
235 : rem* find next prime
240 : for np = 0 to 1
250 :  prime = prime + 1
260 :  np = -(sieve%(prime) = 0)
270 : next np: rem* until zero found
280 :
285 : rem* set multiples of prime
290 : for set = prime to max step prime
300 :  sieve%(set) = 1
310 : next set
320 :
330 : number = number + 1
335 : mloop = -(prime> = max)
```

```
340 next mloop
350 :
351 tme = ti: rem* stop timing
352 print: print
360 print number;" primes generated."
370 print tme/60;" seconds taken."
```

The above program was written so that you can easily understand the process, and modify it if neccessary. If you're too lazy too type the whole thing in, here's a simplified and slightly shorter version. Note that what is gained in brevity is paid for in clarity and versatility.

```
1 rem* sieve of eratosthenes *
2 input "maximum prime";m : dim s%(m + 1) : p = 1 : for k = 0 to 1 : print p
3 for i = 0 to 1 : p = p + 1 : i = 1-s%(p) : next : for s = p to m step p : s%(s) = 1
  : next : k = -(p>=m) : next
```

The disadvantage of using the sieve to generate primes is that the amount of memory available limits the highest prime that can be produced. On the 64, the above routines can go as high as about 19000. (Yes, I tried it. The highest prime was 18979. No, I don't know how long it took). Using a single bit per element, you should theoretically be able to get sixteen times that. A simpler modification would be to change the routine so that it doesn't set the prime locations after it prints the primes. This would leave the array intact so that the list could be printed again without re-setting the elements. (That's the correct approach to take: my sieve routines are a bit non-standard. Also, traditional sieve algorithms start with an array of ones and zero out the factors, but since DIM zeros the array free of charge, doing it this way means we don't have to initialize every element in the array).

I hope you found the above piece (or was it a bit?) interesting, even if it wasn't of *prime* importance.

Quick Note: the use of integer instead of floating point variables results in slower, not faster, execution times

Useless Fact:

A program with a line number zero can be RUN by typing anything beginning with the letters R-U-N, such as: RUNNING AWAY, RUN FAST, etc. Also, if you type R-U-N on a line containing other text, you need not type a colon to delimit the RUN command. And if your fingers occasionally go spastic after typing R-U-N, you need not delete that "sufferin suffix" before hitting Return. However,

if there is no line zero in the program, such antics are rewarded with an "?UNDEF'D STATEMENT ERROR".

Useful Fact:

Yes, some obscure little bugs in BASIC can actually be "features". When documenting GOSUBs in a program, instead of using a REM, as in:

```
GOSUB 10000: REM* INPUT THE DATE
GOSUB 20000: REM* EXECUTE OTHER ROUTINE
GOSUB 30000: REM* ETCETERA, ETCETERA
```

You can fit more comments on the line by leaving out the REM, and following the destination line number with any character, for example:

```
GOSUB 10000 'INPUT THE DATE
GOSUB 20000 'EXECUTE OTHER ROUTINE
GOSUB 20000 'ETCETERA, ETCETERA
```

The apostrophe (') allows remarks beginning with numbers, and makes an attractive REM substitute. This tidy method of annotation works with GOTOs, too.

Built-In Debugging Aid [69, 162]

Here's an idiosyncrasy that can be put to good use. On a BASIC 4.0 machine (40/8032), performing **SYS 53027** from within a program prints the message " in ", followed by the line number in which the SYS is located. The program then continues normal execution. This is an easy way to trace a recalcitrant program: just insert this SYS at various points in the code, and the messages will show what parts of the program are being executed. In a way, it's more handy than a regular TRACE function, since it only traces the parts of the program you're concerned with. A couple of notes about it: No carriage return is printed after the message, and if you execute it from direct mode, a strange line number is printed out (well, what do you expect?). If it wasn't such a well-kept secret, it would look as though the subroutine was purposely designed as a debugging aid.

Easy Disk Directory Pattern Matching [6, 147, 195]

If you want to load a selective directory from a 1541 single disk drive, or from drive 0 of a dual unit, you needn't use the complete syntax:

```
LOAD"$0:pattern",8
```

The command,

```
LOAD"$pattern",8
```

. . .will do the same thing. For example, to see a directory of all programs on drive 0 starting with a 'P', just enter:

```
LOAD"$P*",8
```

This leads to any easy way to load just the disk header and number of blocks free:

```
LOAD"$$",8
```

Poison Line Number [159]

Sometimes a computer can get annoyed for the smallest reasons. Enter the following number on your computer (it works with 4032/8032 and 64):

350800

There's actually a whole range of numbers in the same neighborhood that produce the same effect. Try entering it more than once. Why does it happen? Who knows, maybe it's just an unlucky number.

Closing "Forgotten" Files [65, 184]

Editing with Commodore machines is wonderful compared to others, but it can be annoying when all variables are lost whenever a line number is entered, with or without text. Besides clearing variables, though, the machine forgets about all open files. Suppose you OPEN a sequential file to disk and write to it. You MUST close the file afterwards, but if you did any line editing, deliberately or not, the system will think there are no open files, and won't let you close it. Now, you know perfectly well that the file is indeed open, since the light on the disk drive is on.

In such a situation, here are two ways to close the file:

1) The disk drive will automatically close all files when the command channel is closed. To use this feature, just enter:

```
OPEN1,8,15 : CLOSE1
```

ALL open files will then be closed, courtesy of the disk drive.

2) You can change the number of files open in the operating system. This method allows you to close the first file opened, or the first N files opened, rather than all open files like method 1. The change is done with a single POKE:

```
POKE 152,1 on VIC/64
or POKE 174,1 on PET
```

You can then CLOSE the file as usual. If you wish to re-activate more than one old file, change the value of the POKE accordingly.

SAVE-ing a Range of Memory From BASIC [109, 144, 155, 164]

On a 4032 or 8032, you can always save a range of memory from the monitor, for example:

```
S "0:filename",08,8000,8400
```

. . .would save screen memory out to disk. With the C64, such a feature would be even more desirable, so that the picture currently in the high resolution screen could be SAVEd. The 64 doesn't have a built-in monitor like the 4.0 PETs do, but you can SAVE a range of memory by entering a single line from direct mode! Here it is:

```
sys57812"filename",8: poke193,slo: poke194,shi
: poke174,elo: poke175,ehi: sys62954
```

The variables SLO and SHI are the low and high order of the start address, respectively, and variables ELO and EHI are the low and high end address. (SLO = start AND 255, SHI = start/256, ELO = end AND 255, EHI = end/256)

For example, to save the high resolution screen from 8192 ($2000) to 16192 ($3F40) using the filename "screen" on drive zero, the line would look like this:

```
sys57812"0:screen",8: poke193,0: poke194,32: poke174,64: poke175,63: sys62954
```

The file can then be LOADed as usual, with:

```
LOAD"filename",8,1.
```

Cassette can be used instead of disk if you change the ",8" to ",1" when saving and loading. Remember, if you're loading the file back in from within a program, you have to make sure it only gets loaded on the first RUN. For example, the first line of the program could be:

```
10 if f=0 then f=1:load"filename",8,1
```

WAIT A SECOND! [113, 222] **Jeff Goebel, Georgetown, Ont.**

"If you are ever using cassette files on a 64, it is a good idea to first make sure the PLAY button has been shut off. I always include a line:

```
"PRESS STOP ON CASSETTE PLEASE"
```

. . .and a WAIT 1,16 at the beginning of any of my cassette loaded programs. This will STOP the computer until the STOP button on the tape player is pressed. That way, when I later try to read from a file, the PRESS PLAY prompt will again be displayed, and the user has the option to change tapes or whatever, before pressing PLAY. Actually, I can spiff up the standard PRESS PLAY prompt to be almost anything I want by using my own routine. If I include a PRINT statement like;

```
"PRESS PLAY ON TAPE CASSETTE UNIT"
```

. . .followed by a WAIT 1,16,16, the computer will stop and wait till the play is pressed. I then have time to

```
PRINT "THANK YOU"
    or "SEARCHING FOR DATA"
```

. . .before I open the file. Since the play key is already depressed, the computer's own prompt will not appear, and the data will load as normal.

"Actually, the WAITs are universal; WAIT 1,16 will stop until ALL keys are up on the tape unit and WAIT 1,16,16 will stop and wait until ANY key is pressed on the unit. It doesn't have to be the PLAY key specifically."

Checking for SHIFT, CTRL, and Commodore keys [13, 113, 134, 212]

PEEK(653) will yield the state of these three keys; bit 0 for SHIFT, 1 for the "Commodore Key", 2 for CTRL. Study the following example.

```
10 rem* control key demo *
20 print chr$(8): rem* lock case *
30 :
40 for i=0 to 1 step 0
50 ck=peek(653)
60 if ck=0 then print" - none - ";
70 if ck and 1 then printtab(1)" SHIFT ";
80 if ck and 2 then printtab(8)" Commodore ";
90 if ck and 4 then printtab(20)" CTRL ";
100 print
110 next i
```

As you can see, the state of any or all keys may be examined with a single POKE, and an AND to see which key(s), if any, are being held down. (Holding down the CTRL key also slows the speed of scrolling).

Changing Screen Character Colours [108, 142, 144]

A quick way to change the colour of ALL characters on the C64 screen:

```
10 c=1: b=53281: rem* c is colour, b is border color reg *
20 s=peek(b): poke b,c: poke 648,160: print chr$(147)
   : poke 648,4:poke b,s
```

The above works on 64s with ROM version V2, which sets colour memory to background colour when the screen is cleared. If you have other ROM versions (that set colour memory to character colour), use this line 20:

```
20 poke 646,c:poke 648,160:print chr$(147):poke 648,4
```

The first version won't change the current character colour, it'll just change the colour of all characters on the screen.

It works by telling the operating system that the screen is way up in ROM, so that clearing the screen serves only to set colour memory. The screen page pointer is then set up to its normal default value, 4 (screen at $0400).

Death by Garbage [130, 132, 224]

Delays caused by garbage collection (discarding of unwanted strings by the system) are often a minor annoyance, but sometimes uncollected garbage can be the cause of unexpected crashes! Suppose we wanted to write a program to store data from a sequential file into memory, either to be examined there by a program, or to be written to a new file. The following harmless-looking program should do the trick, right?

```
10 rem* read bytes into memory from seq file *
20 open 8,8,1,"0:lots of data,s,r"
30 bm=4096: rem* start of memory for storage *
40 c=0   : rem* counter *
50 rem— loop —
60 get#8,a$: poke bm+c,asc(a$+chr$(0))
70 c=c+1
80 if st=0 and bm<24576 goto 50
90 rem— endloop —
100 close 8
110 end
```

Bytes are read from the sequential file and POKEd into successive memory locations. The program ends when end-of-file occurs, or memory location 24576 ($6000 in hexadecimal) is reached. When run on a 4032 or 8032, the above program seems to work fine – unless the file is more than about 5000 bytes long. On a long file, the machine will suddenly break into the machine language monitor, or simply halt. Inspection of the data after the crash reveals that it has been totally corrupted. What happened?

It may be obvious to some of you who fully understand the nature of strings in Commodore BASIC, but it may be a surprise to the uninitiated. It occurs because the data being POKEd into memory steps on string storage space. One would think that the 8k of memory between $6000 and $8000 would be more than enough to store the strings; there's just A$, right? Well, the string storage space grows each time a new A$ is read, because a new string is created in memory. Each time a new string is created, the bottom-of-strings pointer decreases (this pointer is at 48-49 in BASIC4, 51-52 in VIC/64). The garbage left over from previous strings won't be collected until this pointer decreases until it equals the top of arrays pointer – in other words, when there's no more free memory. Unfortunately, we want to use the memory between the top of BASIC and variables, and the bottom of strings.

What we really want in the case of the above program is garbage collection after every new byte is read in. That will keep RAM free and safe, since the strings will never grow more than a few bytes (one byte will be required to store A$, and another to store the result of A$ + CHR$(0)). The best way to force a garbage collect is by invoking the FRE function. In the above program, we could insert the statement:

```
75 F = FRE(0)
```

This doesn't slow down the program noticeably, since there are only two bytes of garbage to discard each time. In fact, in any program which reads in many bytes of data from disk, or redefines string variables often with GETs or string expressions, it's a good idea to use the FRE function in every iteration of the loop. If the strings are allowed to pile up until a garbage collect is automatically invoked, there could be a long wait in store, especially in BASIC 2.0 machines. A program user may think the machine has crashed during a long garbage collect, and become quite hostile as he turns off the power after waiting ten minutes.

So be careful when POKEing into "free" memory, and use the FRE function liberally in string-intensive programs. As a more drastic measure, CLR will also do the trick, but of course it must be used with care within programs. There's another lesson here: even if a program works fine when tested with relatively small amounts of data, it may die when worked harder or for longer periods of time. By coincidence, if I'm testing a commercial program and it crashes on me, the first word that usually springs to mind, is "GARBAGE".

Drowning in Garbage! [129, 132, 224] — Elizabeth Deal

Liz writes, "We all know about the elegant screen dazzlers. The other end of the computing stick is:"

```
100 rem  drowning in c64 garbage!
110 rem  by elizabeth deal
120 :
130 rem* set top of basic to $4000 *
140 poke55,0:poke56,64:clr:vi = 53248
150 :
160 rem* hires screen at $2000 *
170 poke vi + 17,peek(vi + 17)or32
180 poke vi + 22,peek(vi + 22)or16
190 poke vi + 24,peek(vi + 24)or8
200 :
210 rem* define a string 200 times
220 ta = 56324: for j = 1 to 200
```

```
230 v$ = chr$(peek(ta)) + v$: next j
240 get i$:if i$ = "" then clr: goto220
250 :
260 rem* exit and restore screen *
270 vi = 53248
280 poke vi + 17,peek(vi + 17)and223
290 poke vi + 22,peek(vi + 22)and239
300 poke vi + 24,peek(vi + 24)and247
```

The wild patterns displayed on the screen are as a result of "garbage" – the string V$ is repeatedly redefined, filling memory, which happens to be video RAM.

Single Disk Copy Program [200] Rick Illes, Milton Ontario

The program that follows allows you to make copies of programs, or SEQ files on 2031/1540/1541 disk drives. Files can be of any kind: BASIC, machine language, or sequential. The only limit is the length of the file to be copied, which depends on how much memory your machine has. As it stands, it will work on upgrade and 4.0 BASIC; if you have a VIC 20 or Commodore 64, change these lines:

```
110 poke 828,peek(55): poke 829,peek(56): ds = 0
120 poke 55,peek(45): poke 56,peek(46) + 1: clr
130 t = peek(55) + 256*peek(46): s = t
300 close1: poke 55,peek(828): poke 56,peek(829):clr
```

The program follows:

```
100 rem* single disk copy: by rick illes
110 poke 828,peek(52): poke 829,peek(53)
120 poke 52,peek(42): poke 53,peek(43) + 1: clr
130 t = peek(52) + 256*peek(53): s = t
140 input "filename";a$
150 input "Prg or Seq (P/S)";t$
160 open 1,8,8,a$ + "," + t$: if ds goto 300
170 print "ok. . ."
180 rem* read the file in *
190 get#1,b$: poke s,asc(b$ + chr$(.)): s = s + 1: if st = . goto190
200 if ds goto 300: rem* disk error (basic 4) *
210 close1: print "[CR]insert copy disk into drive #0"
220 print "then press <space>"
230 get b$: if b$ = "" goto 230
240 get b$: if b$<>" " goto 240
250 open 1,8,8,a$ + "," + t$ + ",w": if ds goto 300
```

```
260 print " ok. . . "
270 rem* write the file out *
280 for i = t to s-1: print#1,chr$(peek(i));: next i
290 if ds = 0 then close1: print " [q] done! "
300 print ds$: close 1: poke 52,peek(828):poke 53,peek(829)
```

Editor's note:
Notice the 'IF ST=.' and 'CHR$(.)' in line 190? This is perfectly acceptable for the value zero. Faster too.

Also note the way that Rick protected memory before storing the bytes from the disk file. He first saves the current top of memory, then sets it to 256 bytes above the top of the BASIC program; that's plenty of space for variables in this case. After the program finishes, it restores the top of memory pointer to their original state. This is a good technique, and it avoids the possibility of "death by garbage", as explained in the piece of the same name. The only thing to watch out for though, is if you want to use the above routine as a subroutine in a larger program: The variables and arrays in that case may need more than the 256 bytes provided above the program. To allocate more variable space, just change the "+1" in line 120 to give more than one 256 byte block. **–T. Ed**

BASIC 4.0 String Bug [129, 130, 224]

Here's a bug, reported by Commodore:

"The bug is a failure to detect 'Out of Memory' error. This can cause corruption of string data or programs if space is running short.

"The bug only occurs in BASIC4 and when there are less than 768 bytes (or 3 times the longest string) free after all variables and arrays have been assigned to a program.

"An example of the bug on a 32k PET:

```
10 DIM A(6330)
20 BUG$ = BUG$ + "W" + ".": PRINT BUG$: GOTO20
```

"The above program will concatenate a string of alternating characters 'W.W.W.W.W'. The 'Out of memory' terminating is correct but the string is corrupted after only a few passes.

Solution? The easiest solution is to trap the 'Out of memory' error from within BASIC:

```
IF FRE(0)<768 THEN PRINT " OUT OF MEMORY ERROR " : STOP.
```

Another solution is preventative medicine: Don't concatenate more than two strings at the same time. Doing the concatenation in two steps, ie:

```
BUG$ = BUG$ + "W" : BUG$ = BUG$ + "."
```

. . .will circumvent the problem.

Intercepting C64 System Error Messages [82, 103, 215, 222]

Elizabeth Deal
Malvern, Pennsylvania

By changing the "Error message link" at $0300–$0301 to point to your own routine, you can change the behaviour of the operating system when it prints messages, including "READY.". Your code should jump to the normal error handling routine after it's finished (normally $E38B). The type of error is indicated by the X register; the value $80 (128 decimal) indicates no error, and causes "READY." to be printed.

With that brief explanation, I'll present a few useful applications that were sent in by Elizabeth Deal from Malvern, PA.

1) If you're tired of seeing ?SYNTAX ERROR you can get rid of the insults: using SUPERMON (or a similar machine language monitor), change the vector at $0300–0301 to point to your code:

```
YOURCODE LDX #$80          ;code for no error
         JMP (SAVEDVEC) ;back to operating system
SAVEDVEC . . .             ;here goes whatever was previously
                           at $0300/0301 (normally $8b, $e3)
```

This will suppress all error messages, but still print READY.

2) A slightly more useful thing might be to print all messages at the top of the screen (second line, actually) to prevent scrolling:

```
YOURCODE TXA               ;x holds error #
         BMI OUT           ;no error
         LDA #$13          ;ascii code for "home"
         JSR $FFD2         ;print it
OUT      JMP (SAVEDVEC) ;remainder of error handling
SAVEDVEC . . .             as above
```

3) Selective handling of errors can be useful. In graphic situations it is particularly annoying to get a ?FILE NOT FOUND ERROR (#4) as well as a flashing disk light. The light is enough, let's get rid of the error message:

```
YOURCODE CPX #4          ;code for ?file. . . error
         BNE OUT         ;continue if other error
         LDX #$80        ;fake no error
OUT      JMP (SAVEDVEC)
SAVEDVEC . . .           as above
```

4) you can suppress printing "READY." to avoid messing up the screen. We won't need to save the existing vector here.

```
YOURCODE TXA
         BMI OUT         ;no error
         JMP $A43A       ;normal error with "READY."
OUT      JMP $A47B       ;skip printing "READY."
```

A combination of points 3 and 4 could be useful, and the latter point could be modified so that "READY." is only suppressed in certain circumstances.

There is one drawback to suppressing "READY.": any action that doesn't send a final linefeed, such as LIST, will finish with the cursor one line too high. Small price to pay!

5) This is just a scratch of the surface. The C64 is a programmer's delight, but it can be a nightmare if the housekeeping isn't good. Intercepting the error routine to clean up house (switch out of high-resolution mode, bring back BASIC, restore normal pointers, kill the sprites and so on) permits nightmare-avoidance. Other uses are possible, though I haven't tried them – for instance, how would you like to POKE (address),-40 ? It's a pain to figure out the two's complement value of -40 to feed to some machine language program; using the error vector might help in that one.

Note to Liz: Your hunch was right – we do like this sort of thing.

C64 RESTORE Key Checking [13, 113, 128, 143, 158, 164]

In last issue's Bits & Pieces, there was a little interrupt-driven machine language program which performed a subroutine whenever a given key was pressed. Well, if you want to use the RESTORE key on the 64, there's an easier way, and it's better: you don't have to change the IRQ vector, so it will work even with IRQ-driven programs.

The RESTORE key is unlike any other key on the keyboard. There is no memory location which can be read to indicate whether or not RESTORE is depressed. Rather, the RESTORE key is connected directly to hardware circuitry which generates an NMI whenever the key is struck sharply (a slow, gentle push won't do the trick).

An NMI, or Non-Maskable Interrupt, is a lot like an IRQ (Interrupt ReQuest), except that it can't be disabled by software. When an NMI occurs, the 64 jumps to the location pointed to by the vector at $0318 and $0319 (792 and 793 decimal) – this vector normally points to $FE47. On the 64, NMIs are used for two purposes: The RESTORE key (to warm-start if the RUNSTOP key is also held down), and for the RS-232 software (an NMI is generated when a character is received on the RS-232 port). The RS-232 routines don't affect detection of the RESTORE key, though. By changing the vector at $0318/9, we can point to our own routine. If this routine transfers control to the usual NMI routine at $FE47 after it's finished, then the interrupt will finish normally and execution will continue from the point where the interrupt occurred.

Detecting the key is incredibly simple. First the NMI vector must be changed to point at our routine. Suppose the routine lives in the cassette buffer, at $033C. The vector could be set up from BASIC like this:

```
POKE 792,60: rem* set nmi low byte to $3c *
POKE 793,3  : rem* . . . high byte to $03  *
```

The code at $033C would perform some action, say, set up certain border and background colours, then JMP to $FE47:

```
033C: A9 00     LDA #0      ;black
033E: 8D 21 D0  STA  $D021 ;background
0341: A9 0B     LDA #11     ;dark grey
0343: 8D 20 D0  STA  $D020 ;border
0346: 4C 47 FE  JMP $FE47 ;normal nmi entry
```

That's all there is to it. Now, whenever the RESTORE key is struck, the colours will be set up. The normal operation of RESTORE is not hindered, since the normal NMI-handler routine at $FE47 will perform a warm start if the RUNSTOP key is depressed.

A Questionable Prompt [159, 207]

Ok, we all know that BASIC's INPUT statement does us favours and displays a question mark as a prompt, free of charge. Well, sometimes the question mark is totally out of place, since the prompt message isn't a question at all, like: PLEASE ENTER YOUR NAME? –looks a bit silly, doesn't it? To kill the question mark on any machine, without using POKEs or anything machine-specific, open a file to the keyboard (device number 0) as follows:

```
100 open 1,0: rem* open file to keyboard *
110 print" Please enter your name: ";: input#1,name$
120 close 1
```

Besides killing the question mark, using INPUT in this way does not send a carriage return after entry, so that a message could be printed on the same line as the prompt. Furthermore, you can reject null entry elegantly by adding the line:

```
115 if a$ = "" then 110
```

This makes the prompt seem to ignore a carriage return without text.

While on the subject of opening keyboard files, it should be noted that the real advantage lies in full-screen editing capability. Instead of entering a string in response to the prompt, you can simply move the cursor to any screen line and press RETURN, reading the contents of that line into the INPUT variable. A good application would be user-entry of multiple fields, such as name, address, etc. The user could cursor around to his heart's content, editing the fields to his satisfaction before pressing RETURN over correct fields. The program would read the fields one at a time, and could exit the INPUT loop when a special end string is received, for example, "EXIT".

Fast BASIC HI-RES Point Plot [138, 175]

Here's a short BASIC subroutine which will plot a point on a bit mapped screen. The variable 'B' must be set to point to the beginning of bit mapped screen memory (normally 8192), and the array 'E(' must be initialized with:

```
FOR I = 0 TO 7:E(I) = 2↑(7-I):NEXT I
```

```
1000 rem* plot a point *
1010 I = b + (yand248)*40 + (yand7) + (xand504)
1020 poke I,peek(I)or e(xand7)
1030 return
```

Fast HI-RES Screen Clear [144]

Clearing the bit mapped screen with POKEs from BASIC can be maddeningly slow. Here's a machine language program to zero 8192 bytes anywhere in memory. It's 16 bytes long and fully relocatable. The following BASIC program puts the machine language into memory and executes it, clearing the bit mapped screen at 8192 ($2000 in hexadecimal).

```
10 rem* clear hi-res screen *
20 data 162,  32, 160,    0, 152, 145, 251, 200,
        208, 251, 230, 252, 202, 208, 246,  96
30 rem* load ml prog into memory *
40 for i=0 to 15: read a: poke828+i,a: next
50 poke251,0: poke252,32: rem* start address *
60 sys828: rem* execute clear routine *
```

Decimal to Hex Conversion Table [96, 116, 145, 191, 207] **Brian Dobbs**

Before you breath a dec-to-hex-programs-have-been-done-to-death sigh, please note: this one produces a neat looking table on a printer, for future reference. Even if you already have such a table, running this program will save you a run to your nearest photocopy machine if you need extra copies. As the program stands, it goes from zero to 255, but the top limit can be changed in line 130.

```
100 rem decimal to hex conversion table
110 rem by brian dobbs-timmins, ontario
120 :
130 max=255: rem* highest value in table
140 open4,4,1: x=0: y=1: k$="---"
150 print#4,spc(25)"decimal to hex conversion table"
160 :
170 rem— main printing loop —
180 d=x: gosub280: rem* convert to hex *
190 if x>9  then k$="--"
200 if x>99 then k$="-"
210 print#4,tab(5);x;k$;h$;
220 x=x+1: y=y+1: if y=6 then y=1: print#4
230 if x<=max goto180
240 print#4: close4: end
250 rem— end loop —
260 :
270 rem *convert dec to hex subroutine*
280 h$="": d=d/4096: for i=1 to 4: d%=d
    : h$=h$+chr$(48+d%-(d%>9)*7): d=16*(d-d%): next
290 return
```

Large Characters on VIC or 64 [115, 175]

The ROM character generator is accessible to BASIC on the VIC and 64. By PEEKing into the character generator ROM, you can duplicate the shape of all 512 characters in magnified form (8 times larger). The following program asks for the desired character set, and the character to be printed. It uses the subroutine starting at line 330 to print a large image of the character, using asterisks as pixels. The available character sets are:

Char Set	Description
0	upper case/graphics
1	upper/lower case
2	reverse upper/graphics
3	reverse upper/lower

For the VIC version, change line 160 as indicated and delete lines 350, 370, and 440 to end. The 64 version needs extra code because its character ROM is hidden under I/O, and it must be expressly switched in and out. Unfortunately, switching out the I/O will crash the machine unless the interrupts are disabled, so that must be done as well. Subroutines handle the ROM switching.

```
100 rem**********************
110 rem* print large characters  *
120 rem* transactor magazine    *
130 rem* written sep'84 -cz      *
140 rem**********************
150 :
160 crom = 13*4096: rem 8*4096 for vic
170 for i = 0to7:e(i) = 2↑(7-i):next i
180 :
190 print chr$(147)
200 for loop = 0 to 1 step 0
210 : input " character set (0-3) " ;set
220 : input "              character " ;c$
230 : print chr$(147)c$
240 : cp = peek(peek(648)*256)*8
245 : rem 1st screen loc gives char #
250 : rom = crom + set*1024 + cp
260 : print
270 : gosub 330: rem* print image *
280 : print
290 next loop
300 :
310 :
320 -subroutines:
```

```
330 rem* translate rom image *
340 for i=0 to 7
350 gosub 460 '* char rom in
360 line=peek(rom+i)
370 gosub 540 '* i/o in
380 for j=0 to 7
390 disp=1:if line and e(j) then disp=2
400 rem* space for 0, '*' for 1 *
410 print mid$(" *",disp,1);
420 next j:print:next i
430 return
450 :
460 rem* switch char rom in *
470 poke 56334,peek(56334)and 254
480 rem* turn interrupts off *
490 poke1,peek(1)and 251
500 rem* enable character set rom *
510 return
530 :
540 rem* re-enable i/o *
550 poke 1,peek(1)or 4
560 rem* turn interrupts on *
570 poke 56334,peek(56334)or 1
580 rem* switch in i/o *
590 return
```

C64 IRQ Reset [11]

You know the problem: you want to disconnect an IRQ–driven program, but a RESTORE will also reset other things like your screen and border colours. Here's an easy way to set the IRQ vector back to its normal entry point of $EA31:

```
poke 781,12: sys 64701
```

or in assembler:

```
ldx #12
jsr $fcbd
```

80 Column Right–Justify [200]

The ultimate one–liner: when there's a bunch of stuff on the screen of your 8032, enter this:

```
for i=1 to 80: ?" S ";: for j=1 to 24: ?" T ": nextj,i
```

(The reverse " T " is an insert). 8000 Series PET/CBM owners. . . try starting the line with:

```
poke 213, 159
```

Quick Note: when using the non–relocating load as in:

load "file",8,1

you can use any non–zero value instead of 1, so you can use ",8,8" to make typing it in a little easier.

C64 Zero Page View [94]

On the PETs, a good way to get a look at what's going on in zero page was to run an interrupt–driven routine which would continously display the contents of zero page on the screen. Well, on the 64, there's an easier way:

```
poke 53272,7   (,23 to get back to normal)
```

This tells the VIC–II video chip to find screen memory at $0000, giving you a dynamic view of what's going on there. If you have V2 ROMs, you'll have to fill colour memory with something other than background colour to see it, or use the ROM change method below.

C64 V2 ROM Colour Memory Fix [27, 85, 128]

If you have a C64, try this: clear the screen, move the cursor down a line or two, then type:

```
poke 1024, 0
```

If you see an '@' on the top left of the screen, then you have ROM versions 1 or 3. Consider yourself lucky; you can freely POKE to screen memory and see the results of your efforts. If you don't see the '@', then you have ROM version 2. With this ROM, the kernal routine which clears a screen line 'cleverly' fills the corresponding colour memory with the background colour. Since background equals foreground the result is a truly clear screen. Furthermore, if you've ever run a program for the 64 or typed in a little screen blitz from a magazine that didn't work, it could be because the author wrote it on a V1 or V3 machine and assumed it would work on any 64.

The solution? If you're willing to forsake the RAM underlying the kernal ROM for this cause, you can correct the foolish behaviour by changing just two bytes. First copy the BASIC and Kernal ROM into the underlying RAM. If you have a Machine Language Monitor with a 'Transfer' command (like Supermon or Micromon), this can be done with these two operations:

```
t a000 bfff a000
t e000 f f f fe000
```

This transfers the contents of the 8K BASIC ROM and the 8K Kernal ROM into RAM. But it gets it *from* ROM. . . why does it not try to put it back in ROM? Because the 64 knows you can't possibly mean that thanks to a chip called a FPLA (Field Programmable Logic Array). This redirects data flow to a logical destination that has been preset by the engineers. And yes, it's fast!

Next switch out the ROM and switch in the RAM by putting a 53 decimal into the bank select register (location 1):

```
BASIC:   poke1, 53
Monitor:  : 0001 35
```

Now, at $E4DA, there is the instruction:

```
                LDA $D021
Change this to: LDA $0286
like this:      : e4db 86 02
```

($0286 holds the current cursor colour)

The kernal will be living out of RAM from now on, but POKEing to the screen will always yield visible characters. Seems like a lot of work for just poking to the screen, especially when you could have merely changed the background colour. But there was another reason for this excercise (of course).

Now that you have all of your 64 operating from RAM, anything can be changed. The spelling of keywords and error messages are fun to modify, but more importantly the ROM routines can be altered. JMP instructions can be re-routed or entire routines can be substituted. Most common is the "BRK instruction insert" for examining the state of the machine at any particular point in a routine. With your favourite dissassembler you simply change the first instruction beyond the last instruction you want executed to a BRK ($00). Now when you cause that particular stretch of code to execute it will stop at the BRK and you can peer around awhile.

Logically you should be able to replace the BRK with the instruction you wiped out and continue executing. Some routines will allow such interruptions but others aren't so tolerant. Most likely you'll need to replace the BRK and start over (perhaps with a BRK somewhere else?).

SYScreeching Off Into Oblivion

On any BASIC 4.0 machine, you can easily enter the monitor with SYS4, right? Well, try it with a quote after the 4 like:

```
SYS 4"
```

What happened? We won't spoil it by giving it away – look up the purpose of location 4 to figure it out.

Disabling RESTORE On C64 [88, 106, 107, 134, 164]

If you don't want someone crashing out of your program with the RUNSTOP/RESTORE sequence, here's an easy way to disable it:

```
poke 792, 193   (,71 gets back to normal)
```

The disable POKE pretty much renders the NMI routine impotent, so RS-232 operations won't work while it's in effect.

Quick Note: 255-x = 256+not(x)

Fast Hi-Res Screen Clear From BASIC [136, 155]

Last issue's Bits & Pieces gave a little machine language routine to quickly clear bit-mapped memory. Since then Nick Sullivan from TPUG magazine showed us this neat trick to accomplish the same thing from BASIC. If you create a large array and then CLR it, BASIC will zero out anything in its path, including hi-res screen memory if it happens to be in the way. If you have a hi-res screen within the limits of BASIC variable space, just put this line at the beginning of your program:

```
clr: f=fre(0):dim a((-65536*(f<0)+f)/5-10): clr
```

That's it! Within a second, the screen will clear. You can't use this trick if your screen memory is at $C000, but at the usual spot at $2000, and with BASIC pointers set up normally, it works like a charm.

In Search Of. . . The Perfect Colour Combination [28, 33, 99, 108, 111, 128]

Looking for the perfect background/border/character colours for programming on the C64 with a 1701/1702 monitor? Try this:

```
poke 53281,0 : poke 53280,11 (press Commodore-2)
```

For the VIC:

```
poke 36879,9 (press CTRL-8)
```

Adjusting the bright/contrast controls to look good with this combination results in an easy-to-look-at screen for hours of programming without fried retinas.

Quick Note: If processing time is critical, you can speed up the CPU by turning off the VIC-II video chip in the c64:

poke 53265,peek(53265) and 239.

Put Mental Notes on Disk (or Tape)! [109, 126, 141, 164]

Ever compose your thoughts idly on the screen of your computer? Or draw a neat picture using graphics symbols while idly talking on the phone? Want to save the screen to disk or tape to bring it in again later? Enough questions, here's what to do. Last issue's Bits & Pieces gave a method to save a range of memory. To save the screen (at $0400 on the C64):

```
sys57812"filename",8: poke193,0: poke194,4: poke174,231:
poke175,7: sys62954
(use ',1,1' for tape)
```

Of course, that'll mess up a bit of the screen: that's the catch. To bring back your screen, just LOAD it any time with:

load"filename",8,1 *or* load"filename",1,1 for tape

With a BASIC 4.0 machine, just use the monitor to save the screen:

```
sys4
s"filename",08,8000,83e7
(,8000,87cf for 80 column machines)
```

Unfortunately you can't save memory above $8000 to tape. Pardon me. . . you *can* save it to tape, you just can't LOAD it back. Commodore never expected anyone to require memory above $8000 to be saved so they used the high bit of the address for something else. In the VIC or 64 this anomaly has been dealt with and whatever that bit does is now separated into its own byte.

Assembler Programming Tip [150]

Branch instructions like BNE, BEQ, BPL, etc. can be a pain when your program grows and the branch can't reach the intended destination any more – the assembler gives a "BRANCH OUT OF RANGE" error. You can get around this problem by branching to a JMP somewhere, but for a short easy way to do long branches, consider this:

```
intended branch : BNE SOMPLC
easy long branch: BEQ *+5: JMP SOMPLC
```

This leaves the intent of the branch clear, and doesn't force you to define a meaningless label somewhere.

One Line Decimal to Binary Conversion [96, 137, 163, 191, 207]

Store the value (0–255) to be converted in 'x', then:

```
z$="": for j=0 to 7: k=x/2: x=int(k): z$=mid$(str$(k<>x),2)+z$
: nextj: printz$
```

The Bleeper [82, 91, 111, 208]

This little noisemaker runs on any PET with CB2 sound:

```
10 poke59467,16: for a = 1 to 255: poke59466,a:
   for b = 1 to 255 step a: poke59464,b: nextb,a
20 print chr$(7)
```

40 Column Wordpro Dump [221]

Here's a small program that will print out a Wordpro-format text file to the screen. It will work with Paperclip, but there will be a few bytes of garbage printed at the beginning as Paperclip stores extra information at the start of its files.

```
10 rem* print a wordpro file to screen
20 rem* 40 column version for 4032/c64
100 input "filename";f$
110 open1,8,0,"0:" + f$
120 b = 1984
130 rem b = 33728 for 4032
150 cc = peek(646): c = 54272: rem* only for 64
160 print "[redacted]"
165 rem 25 cursor downs
170 get#1,a$,a$
175 rem- main loop -
180 for i = b to b + 39
190 get#1,a$: poke i,asc(a$ + chr$(0))
210 poke c + i,cc: rem* only for 64
220 if st then 250
230 next i: print: goto180
240 :
250 close1:end
```

Regain [162, 182] **Lenard Painchaud**

```
5 rem* restore pgm after reset or new *
10 ad = 49152: for i = 0 to 21
20 read d: poke ad + i,d: nexti
30 data 169,  8, 141,  2,  8,  32,  51, 165,  24
40 data 165,  34, 105,  2, 133,  45, 165,  35
50 data 105,  0, 133,  46,  96
60 print "to execute this program, use:"
70 print "  sys";ad;":clr"
```

Lenard writes: "It comes in handy when a program crashes and you can't get your cursor back. Before you can use this program, however, you need a reset switch. When you turn on your computer, load and run the Regain program. Now, when the computer crashes, press the reset switch. That doesn't do the trick though. You then have to type sys 49152. Now you will have your program back. You can change the memory location where the ML program is stored by changing the value of AD in line 10."

Note: To reset your computer, you have to momentarily ground pins 3 on the user port – pin 1 is a ground. Connecting a push button across pins 1 and 3 makes a good reset switch – It can save your program's life! The above program will also bring back a program after a NEW.

Warm Start [113, 134, 158, 164] **Nick Barrowman**
Border Flasher **St. John's, NFLD.**

Nick writes: "This small routine doesn't serve any practical purpose but it is an example of how you can use the main basic program loop vector in the C64 (warm start link at $0302). A more practical purpose is auto–run routines. This routine will change the colour of the screen border whenever <return> is pressed (from BASIC) or when a break or restore is performed. Hope you like it!"

```
10 for a=49152 to 49169: readb: pokea,b: c=c+b: nexta
20 if c<>1779 then print"checksum error!": stop
30 sys49152
40 print"basic warm start flasher activated"
50 data 169,  11, 141,   2,   3, 169, 192, 141,   3
60 data   3,  96, 238,  32, 208,  76, 131, 164,   0
```

Double Width [6, 125, 188, 195] **Brian Dobbs**
Directory Printout **Timmins, Ont.**

The following little program will give you a disk directory in two columns, useful for printing out and putting in the disk sleeve. If sending the directory to the screen, it will appear as a normal directory on a 40 column screen, and double width on an 80 column screen.

```
100 rem** directory double width **
110 rem**      by brian dobbs      **
120 rem**      timmins, ontario    **
130 k=4: rem* k=3 for screen, 4 for printer *
135 r=1: open k,k
140 dr=0: rem* directory drive zero *
150 gosub 220: rem* directory subroutine
```

```
160 close3
170 input " another (y/n) " ;an$
180 if an$<> " y " then end
190 print " insert another disk, press any key "
200 geta$: if a$<> " " then 200
210 goto130
220 n$ = chr$(0): h = 256: open1,8,0, " $ " + mid$(str$(dr),2)
230 get#1,a$,a$
240 get#1,a$,a$,a$,a1$: if st then 290
250 d = asc(a$ + n$) + asc(a1$ + n$)*h: print#k,d;
260 get#1,a$: if a$<> " " thenprint#k,a$;: goto260
270 r = r + 1: if r = 2 then r = 0: print#k: goto240
280 d$ = str$(d): print#k,tab(40);: goto240
290 close1
300 return
```

C64 Easy Disk Status [66] **John Currie, Mississauga Ont.**

This tidy little routine sits in the cassette buffer at 828, and will display the current disk error status when executed. It's very handy, since the C64 has no built–in disk status function.

```
100 rem basic loader for disk status
110 a = 828
120 read b: c = c + b: if b = 256 then 140
130 poke a,b: a = a + 1: goto120
140 if c<>8574 then print " error in data statements " : end
150 print " 'sys 828' returns the current disk status "
160 data 169,   0,  32, 189, 255, 169,  15, 162
170 data   8, 160,  15,  32, 186, 255,  32, 192
180 data 255, 162,  15,  32, 198, 255, 169,   0
190 data 141,  19,   3,  32, 228, 255, 172,  19
200 data   3, 238,  19,   3, 153, 127,   3, 201
210 data  13, 208, 240,  32, 204, 255, 169,  15
220 data  32, 195, 255, 160,   0, 185, 127,   3
230 data 170, 200,  32, 210, 255, 224,  13, 208
240 data 244,  96,   0, 256
```

Bounce 8032 [196]

Here's another one of those useless little special effects. For some reason though, this one can hold your attention for hours (well, minutes maybe). It only runs on 8032's, since it uses the scroll down feature unique to that machine.

```
5 sp=32768: for j=0 to 1 step 0: s=153-128*k: k=1-k
10 for i=1 to rnd(1)*15: printchr$(s); : poke sp+rnd(1)*1000,46: nexti,j
```

Filename Extensions With SHIFTed SPACE [182, 224]

Filename extensions such as .SEQ, .ASM, .OBJ, etc. are useful to indicate file types, but some programmers prefer to use a shifted space instead of a period in the filename. Such a file will be listed in the directory with the extension OUTSIDE the quotes around the filename. To load the file back in, you can specify the filename without the extension, or specify the entire filename (including the shifted space) if greater uniqueness is required. You can also use this method to make "notes" about a file — the note will show up in the directory but need not be entered to load the file in.

Easy Screen Print [91]

A powerful and little-used feature of Commodore BASIC is the ability to use a screen file for INPUT. If you open a screen file and then GET or INPUT from that file, you will read characters directly from the screen starting at the cursor position, and advance the cursor to the next character or INPUT field.

There are all kinds of uses for screen input, but a good application is to convert screen memory character codes to their CBM ASCII equivalents. Such conversions are necessary when printing all text on the screen to a printer. The following line of code will dump an 8032's screen to a Commodore printer with an 80 column margin width.

```
1 open3,3: open4,4: print"■";: for i=1 to 80: get#3,a$
  : print#4,a$;: next: close3: close4
```

For 40 column machines or a printer set for column widths greater than 80, use this version — it prints a carriage return every 40 characters:

```
1 open3,3: open4,4: printchr$(19);: for i=1 to 24
2 for j=1 to 40: get#3,a$: print#4,a$;: next j: print#4," ": next i
  : close3: close4
```

Phone Speller

Some telephone numbers are most easily remembered by the letters on the dial. For example, you can get information on 1985 Volkswagens by calling 1-800-85-VOLKS. Wouldn't it be nice to give your friends a similarly catchy way to remember your number? The following program (it works on any machine)

gives all letter combinations from any phone number (zero and one have no associated letters, so 0 or 1 appears). There are 2,187 combinations for a 7 digit number, so be prepared for a long list. And even if there are no pronounceable words in the list, you can invent acronymns. What better way to spend an afternoon than to find phrases to fit 2,187 acronyms?

```
100 rem* phone speller *
110 rem*    dec84/cz    *
120 :
130 open1,3 :rem 1,4 for printer
140 l$ = "000111abcdefghijklmnoprstuvwxy"
150 :
160 input "phone number";pn$
170 n=len(pn$)
180 dim p(n), n$(n)
190 :
200 for i=1 to n
210 n$(i)=mid$(l$,val(mid$(pn$,i,1))*3+1 ,3):p(i)=1
220 next i
230 rem* n$ holds letter groups for each digit in number  *
240 :
250 for i=1 to 3↑n
260 print#1,i,
270 for c=1 to n: print#1,mid$(n$(c),p(c),1);: next c
    : print#1,chr$(13);
280 carry=1
290 for j=1 to n
300 p(j)=(p(j)+carry): carry=0
310 if p(j)>3 then carry=1:p(j)=1
320 next j,i
```

Assembler Programming Tip #2 [145]

If you've ever looked through someone's machine language program and come across a seemingly useless BIT instruction (eg. BIT $FFA2), or an inexplicable .BYTE $2C, there is a method to his madness.

The BIT instruction doesn't do any harm to memory or CPU registers, it just sets the zero, minus, and overflow flags based on the contents of the given memory location. In some instances, BIT is used almost like a NOP, but with one major difference: the two operand bytes used to specify the memory location are part of the instruction, and so are not executed as instructions if the BIT is executed. If the first byte of the instruction ($2C) is skipped however, you can execute a 2-byte instruction. For example, consider the following assembler code:

```
ENTRY1  .BYTE $2C
ENTRY2  LDX #$FF
```

If a program were to execute the code starting at ENTRY1, the CPU would see a $2C which is a BIT instruction, and interpret the next two bytes (the LDX instruction) as the argument for the BIT — in this case, the CPU would see:

```
BIT $FFA2
```

If the $2C was skipped over and instructions were executed from ENTRY2, the CPU sees the bytes $A2, $FF and interprets the LDX #$FF instruction normally.

Using the above technique allows you to enter a routine with the X register intact, and later enter the routine one byte past the start and have the register changed to something else before the routine does its thing. Of course, any register may be used instead, or any 1 or 2 byte op code can be executed after the $2C.

The technique is explained here in case you come across it in someone else's program, since it's a fairly widely used and accepted 6502 programing practice. Generally though, programmers who use tricks like this enjoy writing obscure code to save a byte or two of memory, and don't care if anyone else can look at the program and understand it. Many programs, including those printed in the Transactor, are designed to be easily read by people, not computers, and should keep away from such brain-twisting exercises. But giving such advice to a hacker is about as effective as advising a kid not to step in puddles on his way home from school.

1541/4040 Write Incompatibility Bug [24]

When the 1541 single disk drive arrived, so did a new buzz word: "write-compatible". At first it seemed that diskettes were completely portable between 4040 and 1541 drives. Then reports of some nasty disk failures started circulating. Here's why.

Every sector on a disk starts with a "synchronizing character", a Header block, another sync character, and then the data stored in that sector. "Physically" it looks something like:

(.... = sync **HHH** = Header **DDD** = Data)

4040: **.....HHHHH.....DDDDDDDDDDDDDD.....**
1541: **.....HHHHH...DDDDDDDDDDDDDD.....**

Notice how the second sync on a 1541 disk is shorter than on the 4040. Now you take a 4040 disk and write on it with a 1541. It becomes:

.....HHHHH...ddddddddddddddddddd.....

But that's OK – the 1541 and the 4040 can still cope. There is still enough of the sync and the data block is still the same "length". However, go back to the 4040 and write to the same sector and:

.....HHHHH...dddDDDDDDDDDDDDDD.....

Blammo! The data block starts with residue data from the 1541 write to the second sync character. The data block is now "too long" and the disk returns Read Error 23: Checksum Error in Data Block

Apparently new 1541's (as of July 84) have been modified to allow write compatibility between all 1541 and 4040 diskettes.

Auto Keywords For The VIC, C64, PET, and CBM [141]

Today we have the contender for the 'two liner' of the year contest. This machine language monster consumes less than the equivalent of two lines of BASIC. It sits in the cassette buffer and will reconfigure every (shifted) letter on your keyboard to produce a keyword. It's IRQ driven, but retains the old IRQ to jump through at the end, so if IRQ driven code is already installed, this program won't bother it. The code also operates in direct mode only, which most can appreciate if INPUT statements are used in your program. And, if it comes down to it, the "\" key on the PET/CBM or the (shifted) pound symbol on the C64/VIC will reset the original IRQ and kill the routine.

Now, considering that there are only 26 letters on the keyboard, how are all the keywords accessed? With the VIC and C64 we have 76 keywords in total, and with the PET/CBM models with BASIC 4.0 we have 91. To battle this problem, a memory location within the routine can be altered to supply you with every keyword. This location defines a "window" over the total set of keywords. You can't get access to all the keywords simultaneously, but you can move the 26 keyword window over any part of the command set (ie. the part you use most). Check out any list of keywords for your optimal window.

As shown, the program will give you the first 26 keywords. Since the first keyword is "END", a shifted-A will print "END". Vary location 683 from 128 to

193 for the PET/CBM, or location 882 from 143 to 193 for the VIC and C64. Lower values will move the window over the error messages.

Note For PET/CBM Users: Reset IRQ before LOADing from disk, then sys(634) to start again. The C64 and VIC do not have this bug, but the PETs sure do. The machine will hang until the STOP key is pressed if any IRQ driven wonder is present during a LOAD.

```
10 rem save "0:keyword pet.bas",8
100 rem ** rte/84 - auto keyword for the pet/cbm
110 for j=634 to 774: read x: poke j,x: ch=ch+x: next
120 if ch<>17758 then print"checksum error": end
130 print"sys(634): rem ** to enable": end
140 data 165, 145, 201,   2, 240,  20, 165, 144
150 data 141,   5,   3, 165, 145, 141,   6,   3
160 data 120, 169, 149, 133, 144, 169,   2, 133
170 data 145,  88,  96, 165,  55, 201, 255, 208
180 data  90, 165, 217, 201,  92, 240,  87, 201
190 data 193,  48,  80, 201, 219,  16,  76,  56
200 data 233, 193, 170, 169,  27,  32, 210, 255
210 data 169, 157,  32, 210, 255, 169, 178, 133
220 data  87, 169, 176, 133,  88, 160,   0, 132
230 data  89, 224,   0, 240,  21, 177,  87,  24
240 data  42, 176,   8, 200, 208, 247, 230,  88
250 data  76, 199,   2, 200, 230,  89, 228,  89
260 data 208, 235, 177,  87, 133,  90,  36,  90
270 data  48,  11,  32, 210, 255, 200, 208, 242
280 data 230,  88,  76, 220,   2,  56, 233, 128
290 data  32, 210, 255, 108,   5,   3, 173,   5
300 data   3, 133, 144, 173,   6,   3, 133, 145
310 data 108,   5,   3,   0,   0
```

```
10 rem save "0:keyword c64.bas",8
100 rem ** rte/84 - auto keyword for the commodore 64
110 for j=828 to 970: read x: poke j,x: ch=ch+x: next
120 if ch<>17162 then print"checksum error": end
130 print"sys(828): rem ** to enable": end
140 data 173,  21,   3, 201,   3, 240,  24, 173
150 data  20,   3, 141, 201,   3, 173,  21,   3
160 data 141, 202,   3, 120, 169,  92, 141,  20
170 data   3, 169,   3, 141,  21,   3,  88,  96
180 data 165,  58, 201, 255, 208,  85, 165, 215
190 data 201, 169, 240,  82, 201, 193,  48,  75
200 data 201, 219,  16,  71,  56, 233, 193, 170
210 data 169,  20,  32, 210, 255, 169, 158, 133
220 data  87, 169, 160, 133,  88, 160,   0, 132
```

```
230 data  89, 224,   0, 240,  21, 177,  87,  24
240 data  42, 176,   8, 200, 208, 247, 230,  88
250 data  76, 137,   3, 200, 230,  89, 228,  89
260 data 208, 235, 177,  87, 133,  90,  36,  90
270 data  48,  11,  32, 210, 255, 200, 208, 242
280 data 230,  88,  76, 158,   3,  56, 233, 128
290 data  32, 210, 255, 108, 201,   3, 173, 201
300 data   3, 141,  20,   3, 173, 202,   3, 141
310 data  21,   3, 108, 201,   3,   0,   0
```

VIC users need only make one change. The number 160 in bold becomes a 192 (Also add 32 to the checksum just for completeness).

VIC/64 Clear Screen Line [144]

There's an easy way to clear any line on the screen right from BASIC:

C-64: POKE 781,line: SYS 59903
VIC-20: POKE 781,line: SYS 60045

where 'line' is the screen line to be cleared, in the range 0 through 24.

Move Screen Line [85]

There's another general ROM routine that can be easily put to good use: this one copies 40 bytes from a specified point on the screen to the current cursor position.

C-64: POKE 780,hi: POKE 172,lo: SYS 59848

Where lo,hi represents the beginning of screen characters to copy (lo + 256*hi must be in the range 0 to 999).

For the **VIC-20**, use SYS 59990

While We're Exploring ROM Routines. . .
The Memory Transfer Subroutine [126, 215]

Often you may wish to move a range of memory, for example to transfer screen or hi-res memory in or out. BASIC is too slow, but you don't need to write a general-purpose memory transfer routine in machine language (although many of you probably have by now, anyway).

The Kernal itself has to move memory around a lot, for example when inserting or deleting BASIC program lines, and there is a memory transfer routine built into ROM in all machines.

Before calling the routine, there are three addresses which must be supplied: source start, source end + 1, and destination *end + 1* (not start + 1). The vital information follows.

	src start	src end + 1	dest end + 1	subrtn entry point
PET (2.0)	$5C	$57	$55	$C2DF
CBM (4.0)	$5C	$57	$55	$B357
C-64	$5F	$5A	$58	$A3BF
VIC-20	$5F	$5A	$58	$C3BF

Cheap Video–Game Dept.

RACER is the concept of John Durko of Toronto, Ont. This has got to be one the simplest game programs around that is so much fun to play. The version below is for BASIC 2.0/4.0 PETs (40 or 80 columns) and is only 13 lines long. If that's too long for you, try this slightly compressed version — you have nothing to lose typing in 4 lines of code!

RACER for 40 or 80 column PETs: (clear screen before running)

```
HB  1 w=10: y=21: t=20-w/2: x=21: n=y: l=32768+y*80
      : for r=1 to l: for i=1 to n: pokes,32
FF  2 printtab(t)"(*"spc(w)"*)": t=t+c: s=l+x: get x$
      : if peek(s)<>32 then sr=r: r=l: goto7
KH  3 pokes,160: k=peek(151): x=x+(k=180)-(k=182)
      : c=sgn(c-2*((t<5)-(t+w>40))): next
MG  7 n=rnd(1)*10: c=int(rnd(1)*3)-1: nextr
      : print"■*** crash! score="sr;sr*y*(20-w)
```

Steer left and right with the 4 and 6 keys (or as indicated with VIC/64/16/+4 versions) to stay within the track as it changes its path. The variables 'W' and 'Y' in line 1 control the width of the track and the screen line that the "car" appears on, respectively. A narrower track makes for a more challenging game, as does a lower screen line (greater value of 'Y'), since you have less time to react to changes in the track.

The "long" version of the program below prompts you for track size and car position, providing defaults. It can also be modified to work on the C-64, VIC-20 (joystick or keyboard), C16 or +4. In fact, this program could be made to work on any machine that runs BASIC; all you have to know is the location which stores the current key pressed, and where screen memory lies.

After you crash, your score is given as two values; the first value indicates the number of "turns" that you survived, and the second is scaled to take into account the track width and car vertical position.

Possible enhancements? Dynamically change the width of the track; change the "speed" of the car by changing its line position from joystick or keyboard controls; put random "obstacles" in the track which must be avoided; write it in machine code!

One last point: if you have an 8032, you can speed up the track by setting the top of a window on a line above the car.

Full-featured RACER for PETs:

```
OE  100 rem "  RACER  -jd/cz
DH  110 print" S ** use 4/6 keys for left/right ** "
IB  120 input" q track width  (1-20)   10 ll ";w
EA  130 input" car position (1-23)   20 ll ";cy
BP  140 sc=32768+80*cy: kbd=151: lf=180: rt=182
        : rem** machine-specific **
EK  150 b=1: e=38: tw=w+4: tl=20-tw/2: cx=20: s=sc: n=cy
NF  160 for i=1 to n: poke s,32: printtab(tl)"(*"spc(w)"*)"
        : s=sc+cx
IJ  170 if tl<b or tl+tw>e then inc=(tl+tw>e)-(tl<b)
CD  180 get z$: if peek(s)<>32 then 220
DG  190 poke s,160: k=peek(kbd): cx=cx+(k=lf)-(k=rt)
LA  200 tl=tl+inc: next: sr=sr+1
KO  210 n=rnd(1)*10: inc=int(rnd(1)*3)-1: goto160
PK  220 print" qq *** you crashed!! ** — score: "
        sr;sr*(40-tw)*cy: print" q run QQQ "
```

C64 mods:

```
   140 sc=1024+40*cy: kbd=56320: lf=123: rt=119
       : rem for joystick
or 140 sc=1024+40*cy: kbd=197: lf=51: rt=0
       : rem keyboard; home/del keys
   165 poke s+54272,1: rem colour memory (white "car")
```

VIC-20 mods:

```
   140 sc=7680+22*cy: kbd=37151: lf=122: rt=118
       : rem joystick up/down
or 140 sc=7680+22*cy: kbd=197: lf=62: rt=7
       : rem home/del keys
   150 b=1: e=20: tw=w+4: tl=11-tw/2: cx=11: s=sc: n=cy
   165 poke s+30720,0: rem black car
```

C16/+4 mods:

```
   140 sc=3072+40*cy: kbd=198: lf=48: rt=51
       : rem* crsr left/right keys
```

NEW facts [146, 162]

Many programs, after loading some machine code and changing BASIC pointers to protect the top of memory, contain the following line of code:

```
NEW : CLR
```

This is dumb for two reasons:

1) The NEW command does a CLR automatically, free of charge. In fact, the CLR routine comes directly after the NEW routine in ROM, and NEW just falls through into CLR.
2) Any statement appearing after a NEW, even on the same line, even in direct mode, WILL NOT BE EXECUTED. Thus, NEW:CLR, NEW:PRINT, and NEW:SYS64738 all do the same thing: a NEW.

So even though using the command NEW:CLR doesn't do any harm to your programs, it sure doesn't do any good. Leave off the CLR and let's put an end to this custom before it's carried on to future generations.

C64 Programming Tip [113, 134, 164]

It is often desirable to be able to halt a machine language program by pressing a key. The usual approach is to use the "check stop key" routine at $FFE1 and check the Z flag to see if the STOP key was pressed. This approach will not work, however, with programs that disable IRQs, since the keyboard isn't being scanned.

Another possibility is to actually scan the keyboard for a specific key by storing the keyboard row number (inverted) in location $DC00, and checking location $DC01 for the value of the desired key. The problem with this approach is that the key must be down when the scan takes place for it to register. To make sure that the key is detected, it would have to be scanned frequently and possibly in several places throughout the program. That can be time-consuming, and using the keyboard in this way also interferes with operation of the joysticks.

A good solution involves using the RESTORE key and NMI vector. Point the NMI vector to a routine which sets a flag (stores a nonzero value in some memory location), then jumps to the normal destination of the NMI vector. Whenever the RESTORE key is struck, it generates an NMI and this flag will be set. The machine language program can check the flag and exit if it is set. That way, no matter what the program is doing when the RESTORE key is hit, it will eventually find out about it when it gets around to checking the flag. When the flag is found to be set, it should be cleared (set to zero) before exiting to prepare for future runs.

Defaults in INPUT Statements [135, 207]

When using INPUT statements it's nice to provide the user with a reasonable default so that he/she can just press return in most cases. Here is a good way to do it:

```
10 input " Drive number  0[3 Lefts] " ;dr
```

After the prompt message comes three spaces, the default, followed by three cursor-lefts. If the default is more than one character long, increase the number of spaces and cursor-lefts accordingly.

An added bonus of this technique is that you can reject invalid entries in a very nice way. For example:

```
20 if dr<0 or dr>1 then print " Q " ;: goto10
```

This will simply ignore any input other than 0 or 1, without any drama.

350800 And Its Relatives [125] — Elizabeth Deal, Malvern PA

The "350800 poison number" mentioned in the BITS and PIECES of the Vol 5, Issue 5 Transactor is indeed a member of a class of neat numbers with high byte 137 (in fixed point format). This is due to a little bug in the PET, VIC and the C64 computers, as well as the APPLE. The bug does NOT exist in the RADIO SHACK computers, nor in the Commodore's B-128, Plus 4 and C-16.

Tracking the story down can be fun, and shows that the culprit is an intended error-exit from the routine that converts numeric characters in the BASIC text to a fixed point number. This routine is used for many things, one of them being BASIC line number entry. If you follow the PET code (below), beginning where it says 'START HERE', you'll see that when a number's high byte exceeds hex $19 (25 decimal) the intent is to abort. But . . . the PET code jumps into the middle of the ON routine. Now when, and only when, your entered number has a high byte of 137 (hex $89) we fall into the trap. We pull an item off the stack and crash on trying to execute the code. By now, the action-address has been mangled. In the upgrade PET, we end up in zero page, $C8 being the remaining byte on the stack. Good fun.

You can look up the details in the Butterfield's maps - the 'perform ON' routine can be your starting point for disassembly. After ON comes the 'get fixed point number' routine which contains the multiply-by-10 code which is pretty long, and not reproduced here. The story ends with the call to the CHRGET routine and the loopback. Here are two disassemblies, one from the Upgrade PET (problems) and one from the B-128 machine (all fixed).

```
;perform ON
c853  20 78 d6  jsr   $d678
c856  48        pha
c857  c9 8d     cmp  #$8d   ;gosub token?
c859  f0 04     beq  $c85f
c85b  c9 89     cmp  #$89   ;goto?. . .a trap into which we fall
c85d  d0 91     bne  $c7f0
c85f  c6 62     dec  $62
c861  d0 04     bne  $c867
c863  68        pla         ;<— herein lies the stack problem.
c864  4c 02 c7  jmp  $c702  ;dispatch command/rts
;
c867  20 70 00  jsr   $0070
c86a  20 73 c8  jsr   $c873
c86d  c9 2c     cmp  #$2c
c86f  f0 ee     beq  $c85f
c871  68        pla
c872  60        rts
;
;--->START HERE - get fixed point number
c873  a2 00     ldx  #$00
c875  86 11     stx  $11
c877  86 12     stx  $12
;big loop - do while characters are numeric
c879  b0 f7     bcs  $c872  ;all done - exit
c87b  e9 2f     sbc  #$2f
c87d  85 03     sta  $03
c87f  a5 12     lda  $12
c881  85 1f     sta  $1f
c883  c9 19     cmp  #$19   ;is the number over 63999?
c885  b0 d4     bcs  $c85b  ;yup. . . get out (into the trap!) **
c887  a5 11     lda  $11
;. . . etc - multiply by 10 routine
c8a7  20 70 00  jsr   $0070  ;chrget - from basic text
c8aa  4c 79 c8  jmp   $c879  ;and loop back
```

Here we have the same thinking, but this time in the B-128. An identical code is in the Plus 4 machine, see Butterfield's Plus 4 map in vol.5, issue 5. It's clear as daylight that the problem has been fixed. Entering a BASIC line: 350800 print "what's the point?" just blows off to SYNTAX ERROR. If you follow this disassembly, you'll see the little fix:

```
;perform ON
f8d2b  20 d6 b4  jsr   $b4d6
f8d2e  48        pha
f8d2f  c9 8d     cmp  #$8d   ;gosub?
```

```
f8d31  f0  07      beq  $8d3a
f8d33  c9  89      cmp  #$89   ;goto?
f8d35  f0  03      beq  $8d3a
f8d37  4c  4f  97  jmp  $974f  ;nope. . .go to Syntax Error
f8d3a  c6  75      dec  $75    ;yes
f8d3c  d0  04      bne  $8d42
f8d3e  68          pla
f8d3f  4c  aa  87  jmp  $87aa
;
f8d42  20  26  ba  jsr  $ba26
f8d45  20  4e  8d  jsr  $8d4e
f8d48  c9  2c      cmp  #$2c
f8d4a  f0  ee      beq  $8d3a
f8d4c  68          pla
f8d4d  60          rts
;
;START HERE - get fixed point number
f8d4e  a2  00      ldx  #$00
f8d50  86  1b      stx  $1b
f8d52  86  1c      stx  $1c


;big loop - while characters are numbers
f8d54  b0  f7      bcs  $8d4d  ;normal exit, not a number
f8d56  e9  2f      sbc  #$2f
f8d58  85  0c      sta  $0c
f8d5a  a5  1c      lda  $1c
f8d5c  85  22      sta  $22
f8d5e  c9  19      cmp  #$19   ;over 63999?
f8d60  b0  d5      bcs  $8d37  ;yes, jump out to Syntax Error
f8d62  a5  1b      lda  $1b
;. . . etc multiply by 10
f8d82  20  26  ba  jsr  $ba26  ;chrget-next character
f8d85  4c  54  8d  jmp  $8d54  ;loop back
```

Tickertape [75, 76, 69, 179, 208] **Dave Smart, Russell, Ont.**

Here's a good little ticker-tape routine — it can be used on any machine, but the 64 version has 'tick-tick' sound efects. The nice thing about this routine is that it can handle strings up to 255 characters long.

Usage notes: just put the string to be scrolled in Q$, and GOSUB 100; you can vary the speed by changing the '65' in line 160; line 105 must be changed for 80 or 22 column screens; the string Q$ is left altered by the routine.

```
100 rem tickertape subroutine-dave smart
105 ln = 40: rem # of columns in screen
110 for l = 1 to ln: q$ = "    " + q$: next
120 for l = 1 to ln: q$ = q$ + "    ": next
130 for l = 1 to len(q$)-ln + 1
150 print mid$(q$,l,ln) " Q " ;
160 for t = 1 to 65: next t,l
170 return
```

Add the following lines for sound effects on the C-64:

```
106 poke 54273,70: poke 54278,249
    : poke54276,17: poke54276,16
140 poke 54296,15: poke 54296,0
```

Debugging Aid Update [69, 125] **R.C. Marcus, Agincourt, Ont.**

Mr. Marcus writes,

"In the Volume 5, Issue 05 Bits & Pieces column, a handy 'Built-in debugging aid' for BASIC 4.0 machines was shown.

I would like to add more information along this line. This is a BASIC routine and is in BASIC ROM of the VIC, 64 and by your listing of the 16/+4 memory map, page 25 of the same issue, it resides there as well.

It is referred to as Print 'IN' routine and resides in the following locations: VIC, 56770; 64, 48578; C16/+4, $A453.

A SYS to the appropriate location will provide this handy feature."

Easy Program UN-NEW After Reset [51, 146, 182]

Last issue's Bits & Pieces presented "REGAIN" to restore your BASIC program after a system reset or a new. If you crash and reset, but don't have REGAIN in memory, you can use this method, sent in by Alan Clooney of Cranbourne, Australia:

```
poke 2050,1: sys 42291: poke 46,peek(35): poke 45,peek(781) + 2: clr
```

If this gives an error message then

```
poke 45,peek(781)-254: poke 46,peek(46) + 1: clr
```

1541 Crash With REL Files [7, 183, 202] — John Menke, Mt Vernon IL

"Failure to properly close a relative file crashes the 1541's DOS. Subsequent disk operations will give unpredictable results, probably damaging other files, the directory, and BAM. Use of the initialize command 'I' only apparently and deceptively sets things right. The DOS will not work correctly after the initialize in this case. It must be reset with 'UJ' or 'U:' or by turning the drive off momentarily."

1541 DOS Wedge Tips — John Menke

"Most of the 1541 wedge commands work in program mode with a minor syntax change; whatever follows @, >, /, ↑ or ← must be in quotation marks. There appears to be a problem only with the % command when used in this way. These commands must be on their own on a separate program line; in some cases they'll work as the last statement on a line. Variables are not recognized as file names by the wedge commands. The following program lines illustrate the three most useful applications which I have found for these commands:

```
10 @"$"
20 <
30 ↑"program name"
```

Line 10 lists the directory, and as usual the space bar stops/continues the listing on the screen. Line 20 reads the disk error status and prints it to the screen. Line 30 is useful in loaders, or in chaining programs."

One-Line Decimal ⇋ Base B [96, 116, 137, 145, 191, 207] — A Hooyer, He Soest, Holland

To convert from decimal value 'D' to base 'B' with output length 'L':

```
8 n$="": for i=1 to l: h=d: d=int(h/b): h=h-d*b
  : n$=chr$(h+48-7*(h>9))+n$: next: return
```

Result in 'N$'. To convert 'N$' from base 'B' to decimal, output in 'D':

```
9 d=0: for i=1 to len(n$): h=asc(mid$(n$,i))-48
  : d=d*b+h+7*(h>9): next: return
```

Examples:

```
From decimal to hex: d=4096: b=16: l=4: gosub 8: print n$
From binary to decimal: n$="1000101": b=2: gosub 9: print d
```

Restore Key Fun [134, 143, 158] **Scott MacLean, Georgetown, Ontario**

Most programmers know that

```
poke 808, 205
```

will disable RUN STOP/RESTORE, but it has other minor side effects, such as disabling the LIST function etc. The RESTORE key is tied directly into the 6510 NMI (Non-Maskable Interrupt) line. When you press it, it jumps through a vector at 792-793 to wherever it goes, does its stuff, and has its fun. It is possible to replace the vector with say, 49152:

```
poke 792, 0: poke 793, 192
```

Ha, now when we press RESTORE, it jumps to location 49152, and starts doing things. Let's put an RTI (Return from Interrupt) instruction there with:

```
poke 49152, 64
```

Now press RESTORE! Neat! Wow! When you press it, nothing happens! Try RUN STOP/RESTORE. Still nothing. Try this:

```
poke 49152, 32: poke 49155, 64: poke 792, 0: poke 793, 192
X=226: Y=252
poke 49153,X: poke 49154,Y
```

Press RESTORE. Or this:

```
X=234: Y=232  Or:  X=34: Y=228
```

Quick Note: The 8250 Dual Drive records files in sectors spread 5 apart as opposed to 3 apart in the 4040, 8050, and 1541.

Screen Save Update [109, 126, 141, 144] **R.C. Marcus, Agincourt, Ont.**

The handy method to save the screen which appeared in the recent issue under Bits and Pieces, titled "Put Mental Notes on Disk (or tape)!", can be indeed a useful tool. It can be used as well on the VIC with the appropriate changes to the Kernel SYS's.

Unfortunately, with the limited screen of the VIC this direct mode entry takes up three to four lines, depending on the length of the filename; plus another for the "SAVING . . . " message, so in all, a possible five lines are taken or 110 character positions. As the screen has only 506, this is a fair amount of space just for the saving instructions.

The attached short program is for the VIC with any memory configuration. It provides screen save with a single command; SYS 828. The program saves up to the second last line of the screen, so by placing the SYS command on the second last line, as instructed by the BASIC loader program, it does not scroll the screen or appear when the screen is recalled. This gives 462 screen positions for message characters.

This program places a machine language program in the tape buffer, so it can only be used in disk–based systems. The filename is built into the routine as "scr(shifted space)ml" and appears in the disk directory as " scr " ml; the ml is the reminder I use to indicate that the load command must include the " ,1 " to indicate a relocating load,

To recall a saved screen, LOAD it in with: LOAD " scr " ,8,1. Putting the LOAD on line 18 will cause the "READY." to appear near the bottom of the screen and not mess up the message.

The routine does the save without the SAVING message to conserve screen space, but will print error messages if they arise. It also saves with the replace option, so be sure to rename your old screen file if you don't want it wiped out by the next one you save.

```
100 rem basic loader for screen save
110 for p=828 to 879 : read a : poke p,a : cs=cs+a : next
120 if cs<>6685 then print " error!..in data " spc(14)
      " statements. " : end
130 print " 'sys828'...on the 2nd " spc(11) " last line to " spc(10)
      " save screen. "
140 data 169,  64,  32, 144, 255, 169,   0, 162
150 data   8, 160,   1,  32, 186, 255, 169,   9
160 data 162, 103, 160,   3,  32, 189, 255, 169
170 data   0, 133, 251, 173, 136,   2, 133, 252
180 data 164, 252, 200, 162, 205, 169, 251,  32
190 data 216, 255,  96,  64,  48,  58,  83,  67
200 data  82, 160,  77,  76
```

+4 and C16 Bits [190] **Elizabeth Deal, Malvern, PA**

These computers are miracles. What follows are some notes I have which may well be unintelligible to the beginners, but can be of use to someone familiar with other Commodore machines. The User's manual is rather complete, so these are just additional comments:

Character strings are handled differently than in previous machines: there is no such thing as pointers into a program. All strings declared inside a program

are copied to RAM (usually hidden under ROM). There are no garbage collection delays. Additionally, new functions can be done with the strings:

1. The first one is assignment statement with MID$ on the left. Yup, you're seeing it correctly. It's now OK to code:

```
MID$(a$,4,2) = "de"
```

This will change whatever was in positions 4 and 5 to be "de". Can you see why strings can't live inside a program? Would be a programming nightmare if they did.

2. INSTR function returns a position of one string within another, so

```
INSTR$("xyz","y")
```

returns 2. You can also specify a starting position for the search. The old code

```
for j = 1 to len(a$): if x$ = mid$(a$,j,1) then nextj
```

is no longer needed, and the INSTR function is instantaneous! Hurray to Commodore.

Tape is incompatible with other CBM machines. The connector is different, but that's the small part. The timing is different. It seems that the writing goes at about half the speed of the previous units. The code to accomplish tape writing and reading is enormous. Reading is particularly difficult because the TED chip functions differently: there is no such thing as detecting a negative transition – all transitions have to and are being detected in software. The screen is turned off to permit 1.7 mhz operation. Still, it is a slow process.

Tape errors are funny. If you happen to position a tape to the very tippy–end of a program you don't want to load, the computer reports BREAK error (#30) and does not go on to look for the program you do want. Using error trapping (TRAP statement exists in the language!) is the way to go in program mode.

Generally, error numbers when used with tape are wrong. You may get a DEVICE NOT PRESENT ERROR, when you think it should be a FILE NOT FOUND ERROR. You invariably get BREAK ERROR when the end–of–tape header has been read in. This would be only a cosmetic nuisance, were it not for the fact that a STOP–key also causes a BREAK error. It's hard to tell one from another

There is lots of RAM in the machine, and one tends to play a lot of hide–and–seek games in finding things. Some clues:

1. Page 4 contains various indirect routines that permit taking bytes from ROM or RAM. These routines are used by BASIC.
2. In page 7, specifically at $7d7 is an equivalent routine for use by the machine code monitor.
3. BASIC PEEK returns a byte from RAM. Machine Language Monitor returns a byte from ROM. MLM normally saves only RAM, you can't save ROM. BASIC SAVE normally saves RAM. LOAD loads bytes into RAM, as you'd expect. Sometimes you may wish to change the defaults. It can be done:

(a) To PEEK ROM from BASIC: modify a routine in page 4 (at $0494) to ignore the store instruction:

```
poke1176,44 : now peek ROM : poke1176,141
```

This is fairly safe, so long as you DO NOT WORK ANY STRINGS BETWEEN THE TWO POKE1176 INSTRUCTIONS. This, for instance, is the only way you can get at the character generator ROM from BASIC, as far as I can tell.

(b) To peek/save ROM from the MLM, set bit 7 in the byte at $7f8. Incidentally, the monitor has a nice feature – ">7f8 80" is all you need to type in, the ">" sets bytes and displays just one line of memory.

The MONITOR is nice, it's almost like SUPERMON. But there is a serious bug – they chopped the TRANSFER command: T with overlapping addresses does not work in one direction – when the destination is higher in memory than the source. The bytes just write one over another and you lose all your work. A cure: first Transfer to another, non-overlapping area, then do a second transfer to where you wanted to go in the first place.

Colour memory, as in the C64, contains the colour codes for the 1000 screen bytes. One difference, bit 7 is the flashing bit. Funny things happen when you load the C64 colour map into, what's now called, screen attributes map in the Plus 4. You get flashing for nothing.

To change the colour attributes from the C64 POKEs into the COLOR statements, you'll need to add one to each value, as the Plus 4 colour numbers are from 1 to 16. POKE values are still 0–15, but there is little reason to use them.

The keyboard is a delight. Perhaps a bit too soft, but easy to use. The ESC sequences are a joy to use. There is even a pause-all-output key: two keys actually – CTL-S, with any other key restarting the output. There is only one problem: if you use CTL-S during a program run, and use a subsequent GET statement, the S will appear to the GET statement as a real input – I think it's a bug – the keyboard buffer isn't cleared, so you'll have to do it yourself.

Programmable function keys are useful. Unlike in the B machine, most of them have a carriage return at the end. I don't like this feature, but it can be easily changed. Unfortunately, the keys are active inside a running program. Watch out here: if you use GET, and the user pushes the DIRECTORY key, all the letters in 'DIRECTORY' including the carriage return will be delivered to the GET! If you don't want it to happen, there is a way to disable the function keys: either set them all to null (" ") inside a program and redefine at the end, or POKE their lengths to zero.

The default colours and luminances for the sixteen colour keys are in RAM. They are in a table in page 1 at $113. You can change them as you wish. Machine code people who love to POKE the stack (auto-run programs!) will have to stay away from this area.

Some of the structured BASIC statements are splendid. For instance, the DO WHILE construct permits you to code a loop that will never execute (FOR-NEXT loops always run at least once, unless you test and skip around). The interpreter seems to be looking ahead, almost like a compiler: it skips the loop and all the loops inside it. EXIT permits leaving a loop early. LOOP UNTIL tests a condition at the end of a loop. What more can we ask?

The character table is half the size of that in the C64. Reverse characters aren't in ROM, they are software-generated. You may have to take this into account if you convert programs from the C64 to the +4/C16 machines. Pointing the character base address is simple. It does require POKEs, a rare event in this machine: using the BASIC method (above) or the monitor transfer command, move the characters to any RAM. Then tell the TED chip about the move: tell location $ff13 the page number of the start of your character definitions, then clear bit 2 at location $ff12. That's all there is to it. Much simpler than in the C64. Incidentally, it is perfectly all right to speak in semi-hex to the BASIC interpreter, hence

```
POKE DEC("FF12"), DEC("71")
```

will tell the chip the character base is at $7000. What's that 1 doing in $71? No connection, nothing. Nothing is a one. I don't know why.

When you play with non-ROM character set, a nasty thing can happen: an exit from the monitor or any error in BASIC resets things only half way back to normal. So the screen becomes a mess. Several solutions: TRAP all errors in BASIC. Do not leave the monitor (guarantees learning machine code by the total immersion method). Hold STOP and push the little reset button. Define a function key to (blindly) type the reverse maneuver to set the character base to the default again. Enjoy the crazy screen sight and push some keys while you do so – it's actually an interesting display.

It is possible to move the screen memory anyplace in RAM. The TED chip needs to be told of the move, of course. $FF14 register is the place to use. However, I know of no way to print on the screen when it's not in the standard location. You can POKE it, you can flip it, you can do all sorts of things with the relocated screen, but no printing. The print command ($FFD2) delivers bytes to the default location and only there. It should be possible, if you must print on a relocated screen, to reroute output to your own routine (page 3 vectors) but I doubt that it's worth the trouble.

The GRAPHIC split screen always splits five lines from the bottom. The bottom five lines are in the text mode, the top is bit-mapped. The constant which controls the split raster line is coded in ROM, hence a bit rough to change. However, there is a link in the interrupt-service code which does permit you to modify the place of the split screen, if you must do it.

Disabling the STOP key is a favourite pastime of many people. It's quite easy on the Plus 4 computer – use a TRAP statement and trap error #30 to resume execution. I know, however, of one situation where the STOP cannot be TRAPped. That is in I/O. Tape LOAD illustrates it quite well, as things are slow: the STOP can be TRAPped after the message "LOADING" would appear, not before. While the computer is searching for a header, it uses another way to test the STOP. It looks directly at the keyboard register in the TED chip, and it never tells BASIC about it. The same is probably true with the serial disk, but it is a bit harder to catch, as things happen faster. A moral: to disable a STOP during I/O use a little machine code, especially if your program uses tape. The whole exercise is almost pointless anyway, as the little reset button lets anyone in. I like that.

B-128, 1541, and 8050 Bits [213] Elizabeth Deal, Malvern, PA

I wish Commodore would reconsider their decision to drop the B-machines. B-128 is a terrific machine. Sure it's hard to program, but it's fun. It has superb BASIC, superb keyboard, 2mhz clock, it's fast and pleasant to use!

There are an assortment of curiosities about the machine itself and some disk drives:

Happy news: On the B128 the files close themselves! When an error condition causes a disk file to remain open, editing a program line makes the disk whirr a bit and a file gets closed. It's incomplete, but it's not a * file anymore. Clever and useful – if you keep the drive door down, of course.

There is a RESTORE <line number> command in BASIC, just as in +4.

```
BLOAD "file name" drive,unit,bank,address
```

loads program files and does not cause BASIC to run from the beginning. A splendid feature.

BASIC programs which have machine code tacked on to the end are difficult to manage. They can be run, but don't try to SAVE'em without first fixing the pointers up. LOADing such programs causes the end-of-program pointer to be set to the byte following the three zeros (ouch!). Editing a line (or just pushing a RETURN over a line) on the screen does the same thing.

There appears to be a bug in the screen editor which can unreverse reversed characters such as home, cursor directions, etc. in quotes. This only happens if you insert characters ANYWHERE on a line containing the control characters and only when you use the INST key. If you use ESC-A/C to enter/cancel the insert mode, then the line is not ruined. Something is wrong in the setting of the insert-flag but you can prevent trouble by pushing RETURN twice over such lines if you have used the INST key. Say it again Sam. . .

There is an "initialize the drive" command in BASIC-128. It is DCLEAR D1 (for drive #1). I don't know why, but I keep thinking it can NEW the disk. Funny name.

The DOS built into the 8050-drives that come in the Protecto package is a fairly advanced version number 2.7 as you can see in the sign-on message.

There has been a change in the way character string functions work. While ASC of a null byte still returns ILLEGAL QUANTITY instead of a zero (as in the PLUS 4), ASC of characters outside the string aren't ignored anymore:

ASC(MID$("ABC",4)) is now ILLEGAL.

Machine Language monitor is a bit rough to use. Some pointers:

1. You can enter the MONITOR by a call, bank15:sys14*4096 does it. You will never exit the monitor, even pressing the reset button doesn't work. This can be useful if you are messing with page zero and rather not exit to BASIC. The exit address is in $f03f8/9. It can be changed.

2. Normal entry is via a break; bank15;poke6,0:sys6 does the job. Exit is nasty, it clears the screen, among other things. Once again, you can change the reset vector to a better setup.

3. Probably the most annoying feature of the resident monitor is that all error commands default to loading and running machine language programs. A pest.

4. The G (go) command is dangerous: do not try to Go to another bank, the crashes are unreal.

5. Do not use the Z–command. It tries to work a co–processor, whatever that is, which isn't there. Consequently we crash.

6. There is a nice little monitor, called EXTRAMON, on the TPUG disk. It has a fairly clean exit to BASIC, as it does not clear the screen. However, do not use the B–exit if you have run a Go command. Most likely you will crash.

7. Crashing has a new twist. Much of the time you don't really crash. The cursor comes back, and things may appear normal. But a closer look reveals, for instance, that your BASIC text is mangled up, the transfer sequences may have funny bytes put in them, and so on. So – like in the old days, shut off, and start from scratch – even when you see the cursor.

If you store a byte in RAM that isn't there and try to read it back, do something between the two operations. A little bit of time (Jim Butterfield says 14 microseconds) are needed for the address to vanish and a real byte to come through. Otherwise the read operation gives a false result. I've been putting three NOP instructions, that's 12 cycles. That may be cutting it too close.

If you have a mismatch–type of error in READing DATA items, the B–machine reports an error about the DATA line itself, rather than the READ statement.

I suspect that there has been an undocumented change in the way IEEE devices function since the 4040. Things that are plugged in but not turned on cause a bus crash. For instance, a printer that is connected but not on will cause a crash if you try to LOAD or SAVE. Incidentally, the same is true in the Plus 4 machine – two 1541 disk drives, one not turned on, will also crash the system. Can anyone explain this?

Another change is that a 1541 and an 8050 drive must have the complete error message read off the drive. You can no longer look at the first value (first byte of the error message) and quit if it's good. The whole thing has to be read in to that last carriage return. Failure to do so causes strange problems which are hard to debug. In the case of relative files, the light continues to flash on the 1541, but not so on the 8050. Non–relative files and/or the 8050 give no clue. So read the whole message. Again, this change hasn't been documented by CBM, as far as I know, but I've seen the trouble ever since the 1541 was born, and now get the same behaviour in the 8050. Life sure was simpler with my old PET and 4040!

The MPS 8050 drive with DOS 2.7 has a bug: if you try to copy a file from one drive to another, and the file already exists on the destination drive, the disk crashes. BASIC COPY command and the monitor's wedge crash in this way.

The only way out is to reset the disk (on/off switch) and then button-reset the computer. The fault doesn't seem to be in the B-machine, since it behaves correctly with the 4040 drive (FILE EXISTS DOS-error). My Upgrade PET also has trouble with the 8050, yet none with the 4040.

Do not trust the writeup of the KERNAL routines in the various guides to the B-computer. Some routines are described correctly, others are copies of the C64 guide and may not function the same. For instance, to read ST, a major task on the B, you must set the carry bit. If you don't, you'll be setting ST to a value in the A-register. Not a very nice thing to do, when you're reading a file.

Due to the zero-page pointers, programs saved from the B 128 do not LIST very well on any other Commodore computer. If you need compatibility, put BASIC higher in memory. PET-type of a setup seems to be the best thing - BASIC at 1025 ($401 in bank 1). The pointer to start of BASIC is in bank 15 at $2c/2d.

Keyword token numbers have been shifting recently. You cannot count on the PRINT USING token, for instance, to be the same in the B machine as on the Plus 4. The standard command (PET vocabulary) numbers are the same, but there is no pattern with the expanded commands. A bit nasty for a program such as LISTER.

Volume 6, Issue 02

C64 Keyboard Joystick Simulation [107]

If you ever need to try out a joystick-driven program but you don't have a joystick plugged in, you can simulate the stick by using the keyboard. The keyboard can be used instead of the joystick in port 2 by holding down the space bar while pressing C,Z,B,M or F1 to perform the functions in the table below. The port 1 joystick can't be simulated in some video games, like those which use the keyboard as well as the joysticks, but in most programs you can get joystick 1 functions by just pressing a single key – refer to the table below.

JOY2:

space + C = left
space + Z = down
space + F1 = up
space + B = right
space + M = fire

JOY 1:

CTRL = left
← = down
1 = up
2 = right
spc = fire

1-Line SEQ file read [6, 114]

You've probably typed in a little sequential file-read program many times. Although utilities such as BASIC AID and POWER have such a feature built in, the utility isn't always installed when you need to look at a file. To save typing in several program lines whenever you wish to view a sequential CBM ASCII file, here's a short program to do it. It will print the file to the screen and stop and close the file when the end is reached. You can just tack this line to the beginning of the program in memory and delete it when you don't need it anymore.

```
1 open8,8,8," filename ":for i=0 to 1: get#8,a$: i=st
  : printa$;: next: close8: end
```

C-64 Character Flash Mode [11, 45, 80, 81]

One of the many features of the plus 4 and 16 machines is a "flash" mode, which operates like reverse on/off, but causes all characters printed in that mode to continuously flash at the rate of the cursor. Flashing is a great way to

highlight important text, signal an error condition, etc. Below is a program to simulate flash mode on the C-64. One of the 16 text colours becomes the "flash" colour; anything printed in the flash colour (default green) or the current background colour will blink at approximately the same speed as the cursor. Line 65 sets up the flash colour as 5 for green – change it to whatever you wish.

```
NN 10 rem* data loader for "flash" *
LI 20 cs=0
CG 30 for i=49152 to 49245:read a:poke i,a
DH 40 cs=cs+a:next i
GK 50 :
HI 60 if cs<>12150 then print"!! data error": end
JI 65 poke 49152+19,5 :rem flash colour=5 (green)
DD 70 sys 49152
AF 80 end
IN 100 :
NA 1000 data 173, 21, 192, 141, 22, 192, 120, 169
KO 1010 data 24, 141, 20, 3, 169, 192, 141, 21
GO 1020 data 3, 88, 96, 5, 0, 20, 0, 0
LI 1030 data 206, 22, 192, 208, 61, 173, 21, 192
ID 1040 data 141, 22, 192, 173, 33, 208, 41, 15
BJ 1050 data 141, 20, 192, 160, 0, 132, 251, 169
NG 1060 data 216, 133, 252, 238, 23, 192, 173, 23
GB 1070 data 192, 41, 1, 170, 177, 251, 41, 15
OH 1080 data 205, 19, 192, 240, 5, 205, 20, 192
GK 1090 data 208, 5, 189, 19, 192, 145, 251, 200
NF 1100 data 208, 234, 230, 252, 165, 252, 201, 220
AL 1110 data 208, 226, 76, 49, 234, 252
```

Plus 4/C16 Pretty Patterns [190, 227]

Here's a short one. Try changing the step value for different effects, and the values of 'B' and 'E' for different sizes.

```
10 graphic1,1 : b=20 : e=190
20 for i=b to e step7 : draw 1,b,i to i,e : next
```

This next one gives a different pattern each time. After a pattern is drawn, press any key for a new one. Try a few – some are pretty incredible. It works by drawing boxes of different sizes rotated at different angles, thanks to the flexible BOX command in BASIC 3.5.

```
100 rem* +4 boxspiral -cz *
110 graphic 1,1: color 1,1
120 x1=0: y1=0: x2=100: y2=100
130 n1=rnd(0)*10: n2=rnd(0)*10
150 for angle=0 to 180 step 5
160 box 1,x1,y1,x2,y2,angle
170 x1=x1+n1: y1=y1+n2
190 next angle
200 rem* run again when key pressed
210 getkey a$: run
```

C-64: Text on a Hi-Res Screen [115, 136, 138]

The hi-res screen is so much more fun than just plain, boring old text. You know, a picture is worth. . . But we work with words and numbers so much that it's sometimes hard to give meaning to a diagram such as a bar graph without words of explanation and numbers for scales. The subroutine below lets you label your creations by displaying a given ASCII character on a hi-res screen. The character must lie in one of the usual character cells (25 down by 40 across). Before calling the routine, specify the column (0 to 24) in 'CY' and the row (0-39) in 'CX'. The character itself must be in the variable 'CC$'. The program copies the eight-byte character definition from ROM into hi-res screen memory addressed at $2000.

```
1000 rem* put text on hi-res screen *
1010 rem* character rom
1020 rom=13*4096+1024*(peek(53272)and2)
1030 c=asc(cc$): print" S ";
1040 rem* convert ascii to screen code
1050 cc=c+64*(c>64andc<192)+128*(c>191)
1060 rem turn off irqs and select character rom
1070 poke56334,peek(56334)and254
1080 poke1,peek(1)and251
1090 rem* copy from character rom to hires screen
1100 br=rom+cc*8: bs=8192+cy*320+cx*8
1110 for i=br to br+7: poke bs,peek(i)
1120 bs=bs+1: next
1130 rem* switch back i/o in place of char rom
1140 poke1,peek(1) or 4
1150 poke56334,peek(56334) or 1
1160 return
```

Subroutine notes:

1) Line 1020 chooses upper/lowercase or uppercase/graphics mode for displaying the character, depending on the current mode.

2) If the hi-res screen is located in memory somewhere other than $2000, change the '8192' in line 1100 to the actual location.

"Someone's coming" or "Boss" mode

For those of us who work with computers as an occupation, it's hard to load up a game for a bit of stress-relief without feeling some guilt. If you work in an office, you may find yourself looking over your shoulder between blasting meanies in space - some stress relief.

To let you play at ease, several games for the IBM PC (which are primarily used for business - no having fun allowed) have a "someone's coming" mode. When you hit the "boss" key, the game instantly disappears from the screen and is replaced by a fake spreadsheet, word processor or bar graph display. When the big guy once again leaves the room, you can continue your game right from where you left off with another strike of the boss button.

Sounds like a good idea. Might be good for the home computer in case you're killing klingons when you should be cutting the grass. When your wife looks in on your progress, just hit the button and, "just a minute dear, have to balance last month's budget first." To cover all bases, maybe every game should have "boss", "spouse", and "parent" functions built in. Well, game developers? How about it?

Fast Key Repeat [44] David Jankowski, Manoora, Australia

David writes:
"I would like to submit this small interrupt-driven routine. Its purpose is to speed up the keyboard repeat (about 74% faster) for game programs that use the GET command to receive instructions. The program sits in the cassette buffer and runs on the C64.

```
5 rem c64 fast key repeat
10 for i=828 to 847: read a: poke i,a: next
20 data 120, 169,  3, 141, 21,   3, 169, 73, 141,  20
30 data   3,  88, 96, 169,  0, 133, 197, 76,  49, 234
```

For the VIC, change the second last value in line 30 (49) to 191.

Modem Speed-Up [177]

Daniel Bingamon of Batavia, Ohio gives this command to speed up a 1600, 1650 or 1660 modem to 450 baud:

```
open 5,2,3,chr$(0) + chr$(0) + chr$(12) + chr$(4)
```

Now you might be asking, "Who would you connect to at 450 baud? Most are either 300 or 1200." Well, the sequence above has been around just long enough for some authors, like Steve Punter, to make provisions in their bulletin board software. Once connected, you have the option of changing to the higher speed.

1200 Baud Fallacy [176]

Have you ever been told that 1200 bps transmission is too fast for normal telephone lines? The Phone Company, and their gullible subscribers, will rhyme off a rather technically believable line like, "the bandwidth of the signal encoding equipment is not wide enough to handle some frequencies at 1200 bits per second - you need a special line installed to avoid dropouts", which costs you more, of course.

Don't believe it! I have talked to bulletin board systems as far as 3000 miles away with absolutely no trouble. In fact, the entire Transactor magazine is sent over a regular garden variety phone at none other than 1200 bps - no sweat.

True, most 1200 baud modems are somewhat overpriced but there are some deals to be had. And once you start downloading at 4 times the speed you're familiar with, you'll be spoiled for life. Actually, 1200 is becoming quite popular. Some systems will even detect your transmission speed at connect time and automatically adjust themselves to suit.

B to PET/CBM Program Converter [19, 169, 213, 216]

A quick B fact. Basic programs SAVEd by a B machine have a start address of $0003 in zero page. The B machine accepts and relocates PET/CBM Basic programs as if they were its own, just as the Vic and C64 do. But just try to LOAD the Basic program back into the PET or CBM. It will destroy zero page, the stack, and whatever else lies in its wake depending on the size. A horrible awakening for PET people, until now. The program listed below will take your Basic B program and relocate it for the PET or CBM. It will re-create a new Basic program on diskette starting at $0401, with each link address also correctly relocated. A pretty terrific utility. Our special thanks to Jack Weaver of Input Systems Inc. in Florida for this one.

```
PH  100 rem ** change b-128 program to run on 80/4032
FL  110 rem ** jack weaver  input systems, inc.
LF  120 rem ** 15600 palmetto lake dr. miami fl 33157
CD  130 rem ** phone (305) 252-1550
AA  140 :
```

```
IA  150 cu$=chr$(145): cd$=chr$(17): cl$=chr$(157):
        ry$=chr$(18): rn$=chr$(146)
FK  160 c$=chr$(0)
KH  170 open 15,8,15,"i0
AM  180 def fn r(x)=(b2*256+b1)+1022
IP  190 if d0
JD  200 print cu$;ry$;"name of b-128 prog to change"rn$"  ";
FA  210 input of$
HO  220 print cd$"new name of the 80/4032 program ";
GB  230 input nf$
AO  240 print "b-128 prg = "ry$;of$: print "80/4032 prg
        = "ry$;nf$
GD  250 print cd$;cd$"OK (y/n) n"cl$;cl$;cl$;
FH  260 input yn$
OF  270 if yn$<>"y" then stop
HM  280 a=1025: open 4,8,4,of$: if ds then print ds$: stop
ED  290 get#4,a$,b$: a1=asc(a$+c$): a2=asc(b$+c$): if a1<>3
        then next: stop
OI  300 open 5,8,5,"@0:"+nf$+",p,w": if ds then print ds$: stop
FJ  310 print#5,chr$(1)chr$(4);
AC  320 if s then 390
HK  330 get#4,a$,b$: b1=asc(a$+c$): b2=asc(b$+c$)
BL  340 x=fn r(x): if x=1022 then s=1: goto 390: rem check for
        end of prg
OD  350 hi=int(x/256): lo=x-hi*256: print ry$;a;rn$;b1;b2;x;lo;hi
FJ  360 a=a+2: print#5,chr$(lo)chr$(hi);
KE  370 if a=x then 320
EC  380 a=a+1
PM  390 if s then print#5,c$;c$;c$;: close5: close4:
        print "converted!": end
PD  400 get#4,a$: s=st: print#5,chr$(asc(a$+c$));
BF  410 goto 370
```

C64 Screen Sizzle **James Cashin, Corner Brook, Nfld**

For the Bits & Pieces obligatory screen blitz, we are proud to present this little two-liner for the 64 from James, alias the 'Happy Hacker':

```
10 poke53280,0: poke53281,1: printchr$(147): poke53281,0:
   poke53272,18: cs=2304
20 for i=0 to 1step0: b=rnd(1)*256: for j=1 to 7: pokecs+j,b: next j,i
```

The program fills the screen with spaces, and continually changes the character definition for a space. If you've never thought that staring at a CRT can be nasty to your eyeballs, try this one out. A starving optometrists' delight!

C64 Simple Banner Program [69, 75, 76, 161] **Jeremy Stewart, North Bay, Ontario**

Here's a short banner program for the C-64 and a printer. The message can be up to 255 characters long, and will reproduce any character including graphics.

```
100 rem**     banner by jeremy stewart     **
101 rem**     for c64 and 80-col printer    **
105 :
110 l = 53248: open1,4
120 as$ = " ********* ": sp$ = "                "
130 rem 9 asterisks, 9 spaces -reduce for shorter characters
135 :
140 input" S input message ";m$: print" S " m$
150 for y = 1024 to 1023 + len(m$): n = peek(y)
160 for z = 1 to 8: a$(z) = " ": next z
170 poke 56334,peek(56334)and254
180 poke 1,peek(1)and251
190 for a = 7 to 0 step-1: b = 2↑a
200 for c = (l + (n*8) + 7)to(l + (n*8))step-1
210 p = peek(c): x = abs(a-8)
220 if (p and b) = b then a$(x) = a$(x) + as$: goto 240
230 a$(x) = a$(x) + sp$
240 next c,a
250 poke 1,peek(1)or4
260 poke 56334,peek(56334)or1
270 for j = 1 to 8: for k = 1 to 4: print#1,a$(j): next k,j
280 next y: close1
```

Notes:
1) Alter the height of the letters as indicated in line 130.
2) Use a character other than asterisks in line 30 for different effects.
3) Change the 'for k = 1 to 4' loop in line 270 for wider or narrower characters.

Break Box Baffler [68, 106] **Tom Johnson, Jefferson, MO**

Here's a terrific bit of code to retard the code buster blues. Written for the Commodore 64, you will find great benefits in locating it high up at $8000 in RAM. This is the area looked at during system reset for the presence of a cartridge. If the correct code is present, ie. CBM 80 etc., then it will be executed. Tom's code takes advantage of this trick. Upon an NMI break in, the code will be executed, thus throwing the 64 into an endless loop. Pretty bad news all packed into 50 bytes.

```
63000 for i = 32768 to 32818 : read j : poke i,j : next i : return
63010 data   9, 128, 216, 128, 195, 194, 205,  56
63020 data  48, 120, 169, 128, 162,   9, 141,   3
63030 data   3, 142,   2,   3, 141,  21,   3, 142
63040 data  20,   3, 141,  23,   3, 142,  22,   3
63050 data 141,  25,   3, 142,  24,   3, 141,  41
63060 data   3, 142,  40,   3, 169,  48, 133,   1
63070 data  76,   9, 128
```

Volume 6, Issue 03

Disk Cleaner [209] **Peter Boisvert, Amherst MA**

"I clean my disk drive read/write head using a diskette–like insert containing a woven cloth disk impregnated with cleaning solution. To clean the head you must insert the diskette and close the door. Now the instructions say to "run the disk drive for 45–60 seconds" by sending any disk command to the drive. I used to use the initialize command. Unfortunately, the disk turns for only 4 seconds or so before it "knocks" the head and stops. To clean the disk properly requires repeating the disk command 10 to 12 times. That's an awful lot of knocking. Since too much knocking can precipitate head alignment problems, I was determined to find a better way. To my surprise the solution was very simple, provided you have a disk map of the ROM:

```
10 rem* 1541 motor spin routine *
20 open 15,8,15
30 rem execute ml at $f97e to start motor
40 print#15,"m-e"chr$(126)chr$(249)
50 for i=1 to 6000: next: rem time delay
60 rem execute ml at $f9e8 to stop motor
70 print#15,"m-e"chr$(232)chr$(249)
80 close 15
```

This short BASIC program executes two disk ROM routines directly, bypassing the 1541 error checking protocol and avoiding the dreaded "knock". Location $F97E in disk ROM is the start of a routine which simply turns the drive motor on, nothing else. Similarly at location $F9E8 a routine exists which shuts off the drive motor. Thus all that is needed is a short program to execute the routines and a delay loop for the cleaning time. When the program is RUN the drive motor turns but the drive LED doesn't light. Ahh, the wonders of direct access programming! The motor will run for a minute and then stop, leaving a shiny disk in its wake. But, make sure the disk drive door is closed when the cleaning diskette is inserted, otherwise the head will not make good contact with the cleaning surface."

Using Peter's technique, here's another 1541 motor spin program that will make it turn whenever the shift key is pressed. You can use SHIFT LOCK to keep the motor running if you wish. This one is handy when working on the drive.

```
10 rem* 1541 motor spin routine #2 *
20 print chr$(147)"hold SHIFT to spin drive motor"
30 print"press CTRL to quit program"
40 open 15,8,15
50 for i=0 to 1
```

```
60 s0 = s1: s1 = (peek(653) = 1)
70 if s1 and not(s0) then print#15, " m-e " chr$(126)chr$(249): rem motor on
80 if not(s1) and s0 then print#15, " m-e " chr$(232)chr$(249): rem motor off
90 i = -(peek(653) = 4): next: rem until ctrl pressed
100 close 15
```

The 1541's amazing " * " [146, 162]

On the 1541, the special filename " * " can be used to load the most recently used file, or if no disk access has yet taken place, the first file on the disk. On other Commodore drives, " * " always loads the first file. If you want the 1541 to behave as the other drives, i.e. you want to load the first program on disk, just use the filename " :* " instead of " * ", for example:

```
LOAD ":*",8
```

World's Simplest Un-Scratch [146, 162, 213]

The " * " filename on the 1541 will let you LOAD the last program SAVEd, even if it has been previously scratched! You probably won't believe it so try it for yourself:

SAVE the current program in memory: SAVE " 0:TEMP " ,8
SCRATCH it from the disk: OPEN 1,8,15, " S0:TEMP "

You may check the directory at this point to make sure it has been scratched.

NEW the program in memory or even reset the C64 with SYS 64738 (don't turn it off and on, as this will also reset the 1541).

LOAD " * " ,8 and your scratched program is back. Now you can safely save it again.

The above technique will not work if you've used any file since the scratched one, or if the drive has been reset. But it's great for those times when you realize you need a file right after you scratch it!

C-64 Directory LOAD & RUN [149, 220, 224] Bob Davis, Salina, Kansas

"The 8032 series have the capability of using shifted RUN/STOP to load and run the first program on disk. . . but the 64 can go one better.

When you save a program, follow the program name with the following four characters:

1) A shifted space
2) Commodore D (The Commodore key and letter 'D' simultaneously)
3) Commodore U
4) Shifted '@'

This will force the disk directory to contain the file name in quotes, followed by ",8:" and all you do is display the directory, move the cursor to the appropriate line and press shifted RUN/STOP to load AND run your program.

While surely someone else has noticed this before, the trick is new to me, and I have not seen it published."

Jumbo Relative Files [7, 163, 183] **Elizabeth Deal, Malvern, PA**

"The B128 and the MPS-80 Drive **can** write large (500k) relative files without a "file too large" error. An old manual (circa 1982) has this incantation for the 8250, which just happens to work on the DOS 2.7 MPS drives:

```
open 1,8,15
xx=0: print#3, "m-w" chr$(164)chr$(67)chr$(1)chr$(x)
close 1
```

Reset, UJ or the above program with xx=255 turns the large-file feature off.

The CBM 8050 test/demo floppy has a program which expands relative files to an 8250 format. It works only on PET 4.0 computers; I don't have one. I find it mildly amusing that the 8050 test/demo wasn't fixed up to work on the B-machine."

APPENDing ML to BASIC [32, 184]

A hybrid program - one using both machine language and BASIC - often consists of a single file on disk containing a BASIC program with machine code tacked onto the end. An easy way to create such a file is to simply SAVE the BASIC part, then send the object from your assembler to the same filename with the ",A" (append) filename extension. For example, using the PAL assembler:

```
100 open 1,8,12,"0:oldfile,p,a":rem append to basic prg file
110 sys700  ;activate "PAL" assembler
120 .opt o1  ;direct object to append file
```

(The PAL example is redundant, since that assembler has hybrid capability, but you can use any assembler, or a BASIC loader program using DATA statements to generate the ML object.)

When using this technique, the assembly origin will have to be set to the end of the BASIC program, which you can find by PEEKing the top-of-BASIC pointers ($2D,2E on VIC/64), and the new pointers will have to be set to the end of the ML object before you SAVE the BASIC (so that variables won't clobber the code). Also, remember that when using an assembler the first two bytes of the ML will be the start address, so you'll have to SYS two bytes past the start to execute the program.

Another Use For ",A" [183]

The filename extension for append (,a) can help out when you're word processing. If you're creating a document and wish to maintain a table of contents, list of references, or any notes that come to mind, you can keep appending to a file by putting a ",s,a" or ",p,a" after the filename (depending on whether you're using SEQ or PRG files). Just set a "range" on the next note you wish to add to the file, and save the range with the above extension. Bits and pieces uses this technique with Superscript to keep a list of B&P authors in a separate file.

Creating DEL Files [65, 126] David Stevenson, Pilot Mound, Man.

"A 'DEL' file may be created as follows:

```
OPEN 2,8,2,"0:TEST,S,W"
OPEN 3,8,3,"0:TEST,S,W"
PRINT#2,"FIRST"
PRINT#3,"SECOND"
CLOSE 2: CLOSE 3
```

The first file opened will become a DEL file. The DOS allows you to open more than one file with the same name as long as you haven't closed any and attempts to recover by giving a different file type designator. If you try this with more than two files all but the first two are lost. To make both files easily accessible just rename, changing the first one in the directory.

This happens with SEQ, PRG or USR files (or a combination) on my 1541. I haven't seen mention of this anywhere."

Neither have we. It seems to work with the 8050 as well.

Read Blocks Free Directly [183]

This will let you directly read the number of blocks free on the current disk without any disk access (the disk must have been previously used in some way).

```
5 rem* read blocks free-1541
6 :
10 lo=250: hi=2: rem $02fa-$02fd
20 z$=chr$(0)
30 open 15,8,15
40 print#15,"m-r"chr$(lo)chr$(hi)chr$(4)
50 get#15,l0$,l1$,h0$,h1$
60 f0=asc(l0$+z$)+256*asc(h0$+z$)
70 print"blocks free:"f0
80 close 15
```

For the 8050 or 8250, make these changes (sorry, no 4040/2040 version):

```
10 lo=157: hi=67: rem $439d-$43a0
90 f1=asc(l1$+z$)+256*asc(h1$+z$)
100 print"blocks free - 0:"f0",1:"f1
```

1541 Track Protect [215] — John R. Menke, Mt. Vernon, IL

It's sometimes useful to be able to reserve certain tracks for later use, or prevent programs and files from being saved to a disk or certain tracks. Here's a short, quick 1541 utility which save-protects an entire disk or designated tracks. It works by writing zeros to the BAM (Block Availability Map), thereby misinforming the DOS that those tracks have already been used and are unavailable.

Conveniently, the BAM is restored and the save-protection removed simply by validating the disk.

```
ON  10 print"save-protect"
EN  20 print"(d) entire disk
IN  30 print"(t) a track
MO  40 geta$:ifa$=""then40
FH  50 if a$="d" then x=4:y=143: goto 100
MD  60 if a$<>"t" then 40
FE  70 input"track number";t
BB  80 if t<1 or t>35 then end
CM  90 x=t*4: y=x+3
CC  100 open 15,8,15
IK  110 open 5,8,5,"#"
```

```
PP  120 print#15,"u1:"5;0;18;0
MO  130 print#15,"b-p:"5;x
MN  140 for i=x to y
LJ  150 print#5,chr$(0);
EK  160 next
FD  170 print#15,"u2:"5;0;18;0
IM  180 print#15,"u;"
GC  190 close 5: close 15
JO  200 print"validate deprotects"
```

Scratch & Save [219] **Bob Hayes, Winnipeg, Man.**

"Unlike SAVE with '@:', this program actually scratches your old file before saving the new one. I initially wrote it as an additional command to the TransBASIC language. Once the program is in memory, type this:

SYS<start address>"filename"

Notice there is no ",8" needed.

Below are BASIC loader and PAL source listings of "Scratch & Save". The start address of these listings is $C000 (49152), but the program is fully relocatable. If you're using a dual drive, you'll have to remove lines 350 and 360 from the source code, and specify the drive number in the filename whenever you call "Scratch & Save".

```
PO  10 rem* data loader for "scratch & save" *
LI  20 cs=0
LF  30 for i=49152 to 49252:read a:poke i,a
DH  40 cs=cs+a:next i
GK  50 :
OC  60 if cs<>14558 then print"* data error ": end
MB  70 rem sys 49152"filename"
AF  80 end
IN  100 :
CB  1000 data  32, 158, 173,  32, 163, 182, 134, 251
BF  1010 data 132, 252,  72, 162,   0, 189,  90, 192
EC  1020 data  32, 210, 255, 232, 224,  11, 208, 245
MB  1030 data 169,   8,  32, 177, 255, 169, 111,  32
PG  1040 data 147, 255, 169,  83,  32, 168, 255, 169
AC  1050 data  58,  32, 168, 255, 104, 170, 160,   0
GH  1060 data 177, 251,  32, 168, 255,  32, 210, 255
MA  1070 data 200, 202, 208, 244, 132, 253,  32, 174
KF  1080 data 255, 165, 253, 166, 251, 164, 252,  32
OO  1090 data 189, 255, 169,   8, 168, 170,  32, 186
```

```
HL 1100 data 255, 169,  43, 166,  45, 164,  46,  76
GN 1110 data 216, 255,  83,  67,  82,  65,  84,  67
HK 1120 data  72,  73,  78,  71,  32
```

```
FD 100 sys700
HC 110 ; scratch and save
LP 120 ; bob hayes; winnipeg, manitoba
NI 130 ; routine help from brian munshaw's
AC 140 ; "new error wedge".
OP 150 .opt oo
MA 160 write  =    *
KB 170        jsr  $ad9e
KA 180        jsr  $b6a3
GD 190        stx  $fb
HE 200        sty  $fc
OE 210        pha
DC 220        ldx  #0
FF 230 mloop  =    *
GD 240        lda  smsg,x
DJ 250        jsr  $ffd2
AO 260        inx
MP 270        cpx  #11
IP 280        bne  mloop
NB 290        lda  #8
FG 300        jsr  $ffb1   ;listen
DM 310        lda  #$6f
IO 320        jsr  $ff93   ;send secondary address
KJ 330        lda  #"s"
PD 340        jsr  $ffa8   ;ciout
DG 350        lda  #":"
DF 360        jsr  $ffa8   ;ciout
KP 370        pla
HE 380        tax
BN 390        ldy  #0
LA 400 sloop  =    *
IN 410        lda  ($fb),y
PI 420        jsr  $ffa8   ;ciout
HE 430        jsr  $ffd2
IJ 440        iny
JH 450        dex
CL 460        bne  sloop
IF 470        sty  $fd
JA 480        jsr  $ffae   ;unlsn
OM 490        lda  $fd
OC 500        ldx  $fb
```

```
PD  510       ldy  $fc
AB  515       jsr  $ffbd    ;setnam
DA  520       lda  #8
BO  530       tay
HO  540       tax
NO  550       jsr  $ffba    ;setlfs (open8,8,8)
BK  560       lda  #$2b
CF  570       ldx  $2d
DG  580       ldy  $2e
PJ  590       jmp  $ffd8    ;save $2b,2c to .x,.y
JJ  600 smsg  .asc "scratching "
```

C-64 POP [9, 10]

Sometimes you need to clean up the stack and re-start a program without killing variables, for example when you need to get back to the main menu from a deeply nested subroutine after an error condition occurs. The POP routine that works on the PET doesn't do the trick for the 64, but you can use this trick instead: just LOAD the program from within itself. That will cause an automatic re-run, cleaning the stack of subroutine return addresses and for..next loops, but leaving variables intact.

C64/VIC20 PRINT AT [210] M. Van Bodegom, St. Albert, Alberta

"On many computers you can move the cursor to any spot on the screen with a simple command. For example, TAB(8,8) or PRINT AT(8,8); would allow you to print starting at row 8, column 8. Commodore doesn't have a BASIC command for this so most programmers PRINT down to the line and then use TAB(column). There is an easy way to get the cursor directly to any spot on the screen. The KERNEL has a routine that does just what we want. Simply use this line to set the cursor location:

```
POKE 781,row: POKE 782,column:
SYS 65520: PRINT " message "
```

Menu Select [147] Tim Buist, Grand Rapids, MI

There have been many menu selection programs, but this is one of the nicest to use, and it's fairly short! Just put the selections in the array 'A$()', the number of choices (up to 11) in 'N', then call this subroutine. It will display the options centred on the screen and highlight the first one. You can use the cursor up/down keys to highlight any option, and confirm the selection by pressing RETURN.

The subroutine returns with the chosen selection number in the variable 'I'. You can then branch the the appropriate section of your main program with ON I GOTO or ON I GOSUB. With the few additions given below, you can select using either the joystick or the keyboard.

```
100 rem* menu subroutine *
110 cd$ = chr$(17): cu$ = chr$(145)
115 hi$ = " r ": off$ = " R "
116 rem use reverse-on and reverse-off for above,
117 rem  any two colours, or a combination.
120 aa = (25-n*2)/2: printchr$(147)
130 for i = 1 to aa: print: next
140 for i = 1 to n: printtab(20-len(a$(i))/2);off$;a$(i): print: next
150 print chr$(19)
160 for i = 1 to aa: print: next: i = 1
170 printtab(20-len(a$(i))/2);hi$;a$(i)
175 get a$
180 if a$<>cd$ and a$<>cu$ and a$<>chr$(13) then 175
190 if a$ = chr$(13) then return
200 printcu$;tab(20-len(a$(i))/2);off$;a$(i)
210 if a$ = cd$ then print: i = i + 1: if i>n then 150
220 if a$ = cu$ then print cu$cu$cu$;: i = i-1: if i<1 then 150
230 goto170
```

Notes:

1) Line 115 is set up to highlight the selected option with reverse field. If you wish, use colours for 'HI$' and 'OFF$', or colours combined with reverse on and reverse off (see comments in program).
2) To allow use of the joystick as well as the keyboard (up/down and fire to select), add the following lines:

```
176 j = peek(56320): rem 56321 for joystick port #1
177 if j = 111 then a$ = chr$(13)
178 if j = 125 then a$ = cd$
179 if j = 126 then a$ = cu$
```

LIST Freeze **Yijun Ding, Pittsburgh, PA**

Here's a real convenience utility. It lets you temporarily halt a program listing in progress to examine a section of code. Saves having to BREAK and re-list all the time! Once activated, this 21-byte machine language demon will live unobtrusively in your C-64 until you hold the SHIFT, CTRL, or Commodore key during a LIST to "freeze" the action. Just RUN the program below to set it up.

```
10 rem* data loader for "list freeze" *
20 cs=0
30 for i=49152 to 49172: read a: poke i,a
40 cs=cs+a: next i
50 :
60 if cs<>2031 then print "!data error!": end
65 sys 49152
70 print "■list freeze activated.
80 print "■press ctrl, shift or commodore keys to
   halt program listings.
90 end
100 :
1000 data 169, 11, 141, 6, 3, 169, 192, 141
1010 data 7, 3, 96, 8, 174, 141, 2, 208
1020 data 251, 40, 76, 26, 167
```

A Couple of Plus/4 Goodies [165, 174, 227]

The first one, **Waving Spokes**, was originally designed to run on a Radio Shack plotter. You'll understand its title when you run it a few times. You can get vastly different patterns by supplying different parameters on start-up. Some recommendations: 20,6,20; 50,4,10; 30,6,60; 40,20,10; 20,4,100

After a pattern is complete, you can press F6 (RUN) to generate a new one.

```
1 rem " waving spokes - plus/4
10 graphic 0,1
20 input "no. of spokes, no. of waves,
   amplitude of waves";spok,waves,amp
30 graphic 1,1
35 p=360/spok
40 for angle=0 to 360-p step p
50 locate 160,100
60 for i=0 to 100 step 5
70 d=amp*sin(i*waves*.01745)
80 x=i*cos((angle+d)*.01745)
90 y=i*sin((angle+d)*.01745)
100 drawto 160+x,100+y
110 next i,angle
```

This next dazzler - **Kaleidoscope** - was originally written for an Atari machine. It's uncomplicated and easy to modify, but produces a constantly changing intricate pattern -- certainly worth a try.

```
1 rem " kaleidoscope - plus/4
50 xm = 159: ym = 199: mc = 1
60 graphic 3,1: color 0,1: color4,1:
   color1,8: color2,2: color3,4
65 do
70 for b = 1 to xm
80 mc = mc + 1: if mc>3 then mc = 1
90 draw mc,b,c to xm-b,c
100 draw mc,b,c to xm-b,ym-c
110 draw mc,b,ym-c to xm-b,ym-c
120 draw mc,b,ym-c to xm-b,c
130 c = c + 6: if c>ym then c = 0
140 next b: color 3,4,i
150 i = (i + 1)and7
160 loop
```

BASIC Programming Tip – Simulated IF..THEN..ELSE

Here is a way you can put a statement on the same line as an IF. . .GOTO and have it execute if the branch *isn't* taken:

```
ON -(condition) GOTO 1000: statement(s)
```

This is equivalent to

```
IF (condition) THEN 1000: ELSE statement(s)
```

Since the C-64 and PET don't have an ELSE, the above trick can come in handy.

See why it works? By negating the condition, we get ON 1 or ON 0, which jumps to the given line if the condition is true, or "falls through" to the next statement if not. A bit tricky, but easier to follow than a rat's nest of GOTOs.

ML Binary/ASCII Conversions [96, 116, 137, 145, 163, 207]

Tim Buist
Grand Rapids, MI

"This first routine is easy to use: just place the binary number you wish to convert after the SYS, for example:

```
SYS 49152, 110010
```

The 16-bit result will be in RESULT and RESULT + 1, which are 828 and 829 in the listing below.

```
100sys700;pal 64 assembler
101;this program converts an ascii
102;binary number to actual binary
103;form and stores it in "RESULT"
104;it works on anything up to 16 bits
105;
110.opt oo
120result   =   828
130        lda  #0          ;clear it first!
140        sta  result      ;lsb
150        sta  result+1    ;msb
160loop    =   *
170        jsr  $0073       ;chrget
180        cmp  #"0"
190        beq  zero
200        cmp  #"1"
210        beq  one
220        rts              ;return if not 0 or 1
230zero    =   *
240        clc
250one     =   *
260        rol  result      ;put in carry bit
270        rol  result+1
280        jmp  loop        ;get more digits
```

While looking like it does nothing, it actually rotates a bit into RESULT. Since a CMP. . .BEQ will sett the carry bit, at ONE the carry bit will be ROLed into RESULT. If the CMP #'0' succeeds, the carry bit is cleared and a zero inserted into RESULT. These Sure are fun to write!

Here's another simple but fun subroutine that converts an 8–bit binary number to ASCII binary and prints it. While this is again a not–so–complicated–that–I–couldn't–think–of–it subroutine, it might spark someone just getting started in M.L."

```
100  sys700;pal 64
101  ;this program converts a byte
102  ;to its ascii binary equivalent
103  ;and prints it.
105  ;
110  .opt oo
120  number  =   828        ;result will go here
130          ldx  #7        ;8 bits
140  loop    =   *
150          lda  #"0"
160          asl  number    ;get a bit from number
```

```
170             adc  #0          ;add in carry
180             jsr  $ffd2       ;print it
190             dex              ;next bit
200             bpl  loop        ;all 8 bits done"?
210             rts
```

Lett'er Fly! [198]

Try this:

```
10 s1$=chr$(19)+chr$(17)+chr$(157): s2$=chr$(19)+
   chr$(29)+chr$(20)
20 get a$
30 print s1$a$s2$: goto 20
```

Press a few letter keys and watch. We know, neat but totally useless, right? Well, modify line 20 like this:

```
20 get a$: if a$="" then 20
```

Now try it. You might have a use for an input routine like that in one of your programs.

Volume 6, Issue 04

Multiple Directory Pattern–Matching [6, 125, 147]

Commodore's filename pattern–matching feature for disk directories is more powerful than many people are aware. One little–used ability is the use of multiple patterns in a directory listing. For example, you could get a list of all files on the disk in drive zero starting with either the letter 'S' or the letter 'D':

```
LOAD"$0:S*,0:D*",8
```

Up to five selective directories may be used in a single directory filename.

Corrupting RAMTAS Routine [220] **Edward Smeda**
Victoria, Australia

RAMTAS ($FF87) is a C–64 Kernal routine which, among other things, has the function of setting the top of memory pointer. This is done by non–destructively testing RAM until it finds a memory location which does not return the value written to it. This location, usually $A000, then becomes the top of memory. RAMTAS is part of the C–64 power–up routine ($FCE2).

Normally, no problems occur with this routine. However, if you have any machine code or other information stored in the RAM under BASIC ROM you will find that a hardware reset (reset button) or software cold–start (SYS 64738) will always corrupt the byte at $A000. This occurs because when RAMTAS tests $A000, it writes the RAM with $55 but, on reading, it reads the BASIC ROM instead and finds a different value. RAMTAS aborts at this point, leaving $55 in the RAM at $A000.

While this does not really qualify as a bug, programmers should be aware that it does occur and should make allowances. There are a number of ways around the problem, but the simplest is to avoid using location $A000 for program or data.

Editor's note: On the other hand, this "feature" can be used to check if a reset occurred since a program was last RUN

Where am I? **Noel Nyman, Seattle, WA**

Relocatable machine language programs are the easiest to use. Invariably some nifty routine from The Transactor sits in a spot needed for another part of your program. it would be best if authors made their code relocatable. This isn't always easy. JMPs within the code are usually necessary and to use JMP commands, absolute addresses are required.

However, if the code can find its own location in memory, the JMP addresses can be calculated regardless of where the user stuck the program.

The "Where am I?" routine below stores a reference to its beginning address before executing the main program. It uses a JSR to force the program counter (the address of the JSR instruction) to the stack, then retrieves the address.

```
        JSR $FFDE ;read real-time clock, or any harmless JSR
        TSX
        DEX
        DEX
        TXS       ;move the stack pointer to the stored address
        PLA
        STA $FD   ;store high byte of address
        PLA
        STA $FC   ;store low byte
(main program)
```

The vector stored at $FC/$FD is the starting address of "Where am I?" plus two. By adding an offset to this value and using indirect JMPs, the program can be made totally relocatable.

QUAKE!! [148]

This is another one of those lovely Transactor specials, frivolous but somehow worth typing in anyway. QUAKE!! will simulate the effect of a 6.0 on the Richter scale, or programming while using hallucinogenics. Good at parties or for practical jokes; amaze your friends! The BASIC loader below will generate the 191 bytes of machine code which unleashes "quake mode" – you'll still be able to program normally while the quake is occurring. Quake mode is activated with SYS 49152 and turned off with SYS 49155. Make sure you have plenty of air-sickness bags nearby!

```
AA | 10 rem* data loader for "quake" *
DK | 11 rem* transactor magazine '85 -cz
KJ | 15 rem save"@0:quake.bas",8
LI | 20 cs=0
KF | 30 for i=49152 to 49342:read a:poke i,a
DH | 40 cs=cs+a:next i
GK | 50 :
FB | 60 if cs<>16666 then print"!data error!":end
DD | 70 sys 49152
EP | 80 rem sys 49155 to stop
KF | 90 end
```

```
IN  100 :
IH  1000 data 76, 49, 192, 76, 112, 192, 0, 0
DA  1010 data 1, 2, 3, 4, 5, 6, 7, 7
PB  1020 data 7, 7, 7, 6, 5, 4, 3, 2
BA  1030 data 1, 0, 0, 0, 4, 5, 6, 7
DD  1040 data 7, 7, 7, 6, 5, 4, 3, 2
FP  1050 data 1, 0, 0, 0, 0, 1, 2, 3
OC  1060 data 4, 120, 169, 88, 141, 20, 3, 169
FG  1070 data 192, 141, 21, 3, 169, 1, 141, 26
NC  1080 data 208, 169, 0, 141, 18, 208, 173, 17
MM  1090 data 208, 41, 119, 141, 17, 208, 173, 22
EJ  1100 data 208, 41, 247, 141, 22, 208, 88, 96
JH  1110 data 173, 25, 208, 41, 1, 240, 11, 169
EF  1120 data 1, 141, 25, 208, 32, 150, 192, 76
KO  1130 data 49, 234, 104, 168, 104, 170, 104, 64
BP  1140 data 120, 169, 128, 141, 26, 208, 169, 49
LH  1150 data 141, 20, 3, 169, 234, 141, 21, 3
MK  1160 data 173, 22, 208, 41, 240, 9, 8, 141
AK  1170 data 22, 208, 173, 17, 208, 41, 240, 9
PO  1180 data 11, 141, 17, 208, 88, 96, 174, 6
CN  1190 data 192, 173, 22, 208, 41, 248, 29, 7
DG  1200 data 192, 141, 22, 208, 173, 17, 208, 41
HF  1210 data 248, 29, 28, 192, 141, 17, 208, 238
PK  1220 data 6, 192, 173, 6, 192, 201, 21, 144
PG  1230 data 5, 169, 0, 141, 6, 192, 96
```

The Schizophrenic Sprite [112]

The shape of any C64 sprite is completely determined by 63 bytes in memory. To change the shape of a sprite, the sprite definitions are usually kept static, and pointers are changed to point to definitions elsewhere in memory. What about doing the opposite – keeping the sprite pointer constant but changing the 63 bytes defining the sprite? What if a sprite definition occurs in screen memory? To find out, enter this short bit of code:

```
10 rem schizo-sprite, cz85
20 vic = 53248: rem vic chip at $d000
30 poke vic,25 : poke vic + 1,100
40 poke vic + 21,1: poke vic + 39,1
50 poke vic + 23,1: poke vic + 29,1
60 poke 2040,16
```

A double-sized white sprite appears, whose shape changes depending on the first 63 characters on the screen – the top screen line and part of the second. The fun part comes by playing. Try different groups of characters:

"ioioioioi"..etc produces the effect of three parallel ladders; repeating the asterisk and english pound characters displays a repeating checkerboard effect; "cxcxcxcxcxcxcxc" is pretty interesting, too (all of these were found by experimenting). Type in your name to see what it "looks like". As usual, we leave it to you to find an application for the above bit of foolishness.

Try This [193]

```
10 geta$:if a$ = " " then printb$;: goto 10
20 b$ = b$ + a$: printb$;: goto 10
```

Press a few keys, then try some cursor controls. It will eventually die with a ?STRING TOO LONG, but by then you'll be tired of it anyway.

Error–Driven Catalog Routine for VIC/64 [82, 103, 133, 215]

This machine–language program sits in the cassette buffer and displays a directory of drive zero whenever a ">" (greater–than) is entered. It works by trapping the syntax–error vector, so it won't bother anyone when it's not in use.

```
LB  10 rem save"0:errcat 64.bas",8
MM  100 rem ** rte/85 – error vector driven catalog
        routine for c64 and vic 20
NJ  110 rem ** press > then (return) for a catalog of drive zero
HG  120 for j=828 to 951: read x: poke j,x: next
DD  130 sys(828)
KK  140 rem
PE  150 data 169, 71, 141, 0, 3, 169, 3, 141
JN  160 data 1, 3, 96, 201, 49, 208, 104, 169
GK  170 data 2, 162, 182, 160, 3, 32, 189, 255
FC  180 data 169, 2, 162, 8, 160, 0, 32, 186
BD  190 data 255, 32, 192, 255, 162, 2, 32, 198
ID  200 data 255, 169, 13, 32, 210, 255, 32, 207
GB  210 data 255, 32, 207, 255, 160, 2, 32, 207
CB  220 data 255, 32, 207, 255, 32, 207, 255, 170
DD  230 data 32, 207, 255, 132, 251, 32, 205, 189
IH  235 rem ok, ok, ok, ok, ok, ok, ok, 221
    Note: use line 235 to change line 230 for vic 20
LG  240 data 164, 251, 169, 32, 32, 210, 255, 32
AC  250 data 207, 255, 32, 210, 255, 32, 183, 255
LI  260 data 208, 19, 200, 192, 28, 208, 240, 32
CD  270 data 225, 255, 240, 9, 169, 13, 32, 210
```

```
KN | 280 data 255, 160,   0, 240, 201, 169,   2,  32
PK | 290 data 195, 255,  32, 204, 255, 162, 128,  76
ML | 300 data 139, 227,  36,  48
JJ | 305 rem  58, 196,  ok,  ok
   | Note: use line 305 to change line 300 for vic 20
```

Notes On REVCNT:
The Error Recovery Count Variable – CBM Drives [200]

Your drive can tell you quite simply when it is out of alignment. By writing a value of 193 to location REVCNT (see below), your drive will err out immediately when an alignment error occurs. The code and an explanation follows below:

```
1541/2031LP :  print#15," m-w " chr$(106)chr$(0)chr$(1)chr$(193)
               : rem loc $006a
2040/4040   :  print#15," m-w " chr$(252)chr$(67)chr$(1)chr$(193)
               : rem loc $43fc
8050/8250   :  print#15," m-w " chr$(245)chr$(16)chr$(1)chr$(193)
               : rem loc $10f5
```

The Reasons Behind Choosing The Value 193 (Binary 11000001)

A quick note on the 6502 BIT instruction. When a BIT is performed on a memory location, the NEGATION flag is set from bit 7 of the location, and the OVERFLOW flag is set from bit 6 of the location.

A BIT instruction is performed on REVCNT by DOS for two different reasons. First, after a BIT on REVCNT, a BVS is made that branches past a routine that executes a track offset. Second, after a BIT on REVCNT, a BPL is made that branches past a routine that tosses a BUMP onto the job que. These two reasons explain why Bits 7 and 6 were set (192), but still leaves the last bit, Bit 0, unexplained. Look below for the answer.

Whenever an error occurs when reading or writing to disk, the routine is attempted a set number of times before aborting. Location REVCNT holds the key to the number of attempts. The DOS will AND location REVCNT with #$3F, storing the result in the Y register for a counter of the number of attempts. If you were to AND 192 with $3F, the result would be zero:

```
11000000 (192)
00111111 ($3F)
--------
00000000 after ANDing
```

Therefore, in order to not loop through 255 cycles of attempts (DEY, BNE routine), bit 0 has to be set. This gives a total value of 193 (Bits 7, 6, and 0 set)

Original 1541 tip thanks to the Central Coast Commodore Users Group Newsletter – April 2, 1985.

ML Right Justify [141] **Richard Perrit, South Porcupine, Ont.**

In Volume 5, Issue 6 we ran this one-line "right justify" for 80-column computers:

for i = 1 to 80: print" [S]";: for j = 1 to 24: print" [T] " : next: next

Richard Perrit of South Porcupine, Ontario has since re-written this special effect in machine language. The program is relocatable and can be installed using the BASIC loader below.

```
CK  10 rem *** right justify 80 ***
GD  20 rem *** richard perrit  ***
LO  30 rem *** august 11/85  ***
MJ  40 :
JE  50 rem ad=49152 for c-64
GE  60 rem ad=634  for pet
JL  70 rem must have 80 columns
EM  80 :
IL  110 ad=634: for i=ad to ad+31: readx: ch=ch+x:
    pokei,x: next
EM  120 if ch<>4605 then print" !data error! " : stop
FK  140 data 169,   0, 162,   1, 160,   1, 169,  19
IO  150 data  32, 210, 255, 169, 148,  32, 210, 255
CO  160 data 169, 141,  32, 210, 255, 200, 192,  24
AB  170 data 144, 241, 232, 224,  80, 144, 229,  96
```

Slipped Disks: [24, 131, 199] **Scott Maclean,**
Speeding up your disk drive **Georgetown, Ont.**

This item deals with speeding up dual drives – examples are given for the 4040, 8050 and 8250. Unfortunately, the method given here will not work on the 1541, because the method we are using does not exist on the 1541.

In the dual drive memory map, at location $1000 (4096 decimal), to location $1003 (4099 decimal) are 3 interesting variables. (Note: 8250 values also apply to the 8050 drives)

Location		Contents (4040)		Contents (8250)		Label	Description
Hex	Dec	Hex	Dec	Hex	Dec		
$1000	4096	$0A	10	$03	3	ID	Interrupt Delay
$1001	4097	$0D	13	$0D	13	MAD	Motor acceleration delay
$1002	4098	$30	48	$30	48	MCT	Motor cutoff time

We can change the contents of these locations to change the speeds of the different functions of the disk unit. We can change the value of the Interrupt Delay, which increases or decreases the overall speed of the drive, including the transfer rate of the drive. Very small delay rates will cause read errors and the drive won't read a thing from disk. The most noticeable thing this value changes is the speed at which a "drive bump" occurs. For instance, set this to 5 on a 4040 and then open a file to disk with the drive door open to cause an error. You will hear a buzzing noise instead of the familiar "WHAPWHAPWHAP" noise a 4040 makes. Also affected is the stepping rate, if you send the head from track 1 to track 35, you will notice a significant increase in stepping speed. A safe value for the 4040 is 9, and for the 8050/8250 is 2.

We can also change the Motor Acceleration Delay rate. When you tell the drive to access the disk, it turns on the drive motor, then waits for a certain amount of time for it to accelerate and stabilize to exactly 300 RPM. We can change this value to change how long the startup delay is. Safe values for all drive types is 2. This value has the most visible effect, as it decreases directory search times, and generally speeds all internal disk access up. Using these two functions, you can read the directory from a 4040 with about 1 second of drive motor time. After setting these two locations and requesting a directory, the 4040 will do a drive bump, move to track 18 and seem to stop instantly. However, it will continue sending directory data until it has finished the directory.

The last location is the Motor Cutoff Time. This is the delay the drive uses after a file is closed, or after data stops flowing. Normally, after you finish using the drive, it will whirr for a few seconds longer, even though it isn't doing anything. By changing the value in this location you can control how long it will continue to spin the disk. If you are used to the length the 4040 spins, and you then start to use an 8250, you will notice that the 8250 seems to take forever to stop spinning. Using all three locations, it is possible to change the entire speed characteristics of the drive. Following is a table showing the safe values for each location, followed by a short program that can be used to change the values easily and quickly.

One last note: I would expect that the same method should operate correctly on the SFD-1001, but don't quote me on that as I have never used one of those units.

Location		Lower Limit		Upper Limit	
Hex	Dec	4040	8050/8250	4040	8050/8250
$1000	4096	$0A 10	$03 3	$F5 250	$F7 252
$1001	4097	$02 2	$02 2	$FE 254	$FE 254
$1002	4098	$02 2	$02 2	$FE 254	$FE 254

Editor's Note: The above Lower Limit values may not work on all drives - experiment. Also, speeding up your drive may make it less reliable; don't trust important data or complex disk functions to a hyped-up machine.

```
10 rem **program to change velocity
20 rem **values of dual drives
30 rem **by scott maclean
40 open 1,8,15:rem **open command channel
50 print chr$(147)
60 input "Interrupt Delay";id
70 input "Motor Accel. Delay";mad
80 input "Motor Cutoff Time";mct
90 print#1,"uj":rem **reset drive
100 print#1,"m-w"chr$(0)chr$(16)chr$(3)chr$(id)chr$(mad)chr$(mct)
110 rem **sets up at locations $1000-$1003
120 close 1
```

```
10 rem **quick program to speed up
20 rem **dual drives
30 open 1,8,15:rem **open command channel
40 print#1,"uj":rem **reset drive
50 print#1,"m-w"chr$(1)chr$(16)chr$(2)chr$(2)chr$(2)
60 close 1
```

1541ders [7, 163, 183] **Daniel Bingamon, Batavia, Ohio**

When I attempt to open a relative file with a record length of 58 (ASCII code for colon) I get errors. It appears that the 1541 likes to think of the colon as a delimiter and since between the comma and the colon is nothing, you get an error for opening a file of record length zero. Maybe this will give Commodore the hint to tear into their source and fix this along with a few other problems (like SAVE@), if we find enough bugs.

The "UJ" command sent via the command channel is being used by some widely sold software. Some drives (most of them) require three seconds for the reset, but some software only waits one second or less. this causes the computer to "hang up" when further disk commands are given. This can occur when the programmer writes a routine in BASIC, then compiles and does not compensate for the speed increase in the FOR..NEXT time delay loops.

"STP" stands for "Sequential To Program". This is a BASIC STP for those who don't want to STP the M/L way. Refer to Chris Zamara's STP program in Transactor Vol 5, Issue 6.

This routine will enter any program that has been listed to a SEQ file on disk. It uses the Dynamic Keyboard technique from BASIC.

As a dividend, BASIC STP may be used to append or merge several programs together. The individual program lines must have no duplicate numbers or your final program will be a total mess.

A great idea is to have a series of routines, with specific numbering for each category of routine. Call and merge them together with BASIC STP. Build a program of routines, using BASIC STP to do it.

To use it for appending a program or routine to an existing program, you may LOAD Basic STP and list it to the screen. Then LOAD the program you are using as the "master" program. Bring the cursor up to the top line of BASIC STP, and hit RETURN over all the lines, 63990 through line 63999. Now BASIC STP is appended to the program.

RUN 63990, and enter the file name of the routine or program on SEQ file you wish to append or merge with your "master" program. BASIC STP will do just that.

The last step is to delete BASIC STP lines, and SAVE the new program.

```
KK  63990 poke828,169: poke829,0: poke830,76
FM  63991 poke831,49: poke832,243: close4
PG  63992 input"filename";f$: open4,8,4,f$: get#4,a$,a$:
          poke829,1: a$=""
MP  63993 print"Saqapoke812,60: poke813,3aq": if a$<>" "
          then 63995
PI  63994 get#4,a$
FI  63995 printa$;: if a$<>chr$(13) then 63994
IO  63996 get#4,a$: a=0: if st=0 then a=asc(a$+chr$(0))
NB  63997 print"a$=chr$("a"): goto63993
BK  63998 if st then poke829,0: close4: stop
GO  63999 poke198,3: poke631,13: poke632,13: poke633,13:
          print"s": end
```

Gaussian Elimination Routine **Audrys Vilkas, Goleta, CA**

The following routine is capable of solving up to nine equations in nine unknowns of the form Ax = b. It can also solve or yield information about non-square arrays. It is done entirely off-screen but the user should be aware that a little gentleness in key input is appropriate. The routine occupies 700 or so odd bytes in the raw and is an excellent tutorial for those who study matrix theory.

```
EL  100 rem * gaussian elimination routine *
FI  110 print: input "Row Dimension";n: input "Column
        Dimension";m
LO  120 dim a%(n,m+1),b(n+m+1): for i=1 to n: for j=1 to m+1:
        k=i+j
MN  130 print"a";i;j;: input" =";a(i,j,b(k))
AP  140 print" Q";: next: next: print: print" rNext Row Dim";n-1;
        "Next Col Dim";m-1
NG  150 for i=1 to n: for j=1 to m+1: printa(i,j,b(k));: next: print:
        next: print
PP  160 for i=1 to n: for j=1 to m+1: def fna(i)=-a(i-1,1,b(k))*
        a(i,j,b(k))
KK  170 def fnb(i)=a(i,1,b(k))*a(i-1,j,b(k)): r=fna(i)+fnb(i)
KA  180 r1=-a(i-1,1,b(k))*a(i,j,b(k)): r2=a(i,1,b(k))*a(i-1,j,b(k)):
        ra=r1+r2
CD  190 r1=a(i,1,b(k))*a(i-1,j,b(k)): r2=a(i,1,b(k))*a(i-1,j,b(k))
        : rb=r1+r2: r=ra*rb
GN  200 r=fna(i+1)+fnb(i+1): printr;: next: print: next: if m=1 and
        n=1 then 220
PP  210 clr: goto110
PH  220 y=a(n,m+1,b(k))/a(n,m,b(k)): print y;" is a solution": clr:
        goto 110
```

The Lottery Companion [119]

When you run out of birthdates, license numbers and hats to pull numbers from, you might want to use this program the next time you play a lottery. It will pick up to ten sets of six numbers, chosen from a pot of 39 or 49, as you choose.

```
OO  100 rem save"0:lottery",8
KA  105 rem ** an evers co-production 1985 **
MJ  110 dim win%(49,10), out$(10): c$=chr$(147)
JG  115 print c$"select option"
DF  120 print "1) lottario 6/39"
HD  125 print "2) lotto  6/49"
NF  130 input x$: if x$<"1" or x$>"2" then 130
```

```
CF  135 lot=39: if x$="2" then lot=49
IN  140 input " output (3) screen (4) printer ";dv
        : if dv<3 or dv>4 then 140
IG  145 open 1,(dv)
FA  150 input " required # sets (1-10) = ";max
        : if max<1 or max>10 then 150
HN  155 print#1," your 6/" mid$(str$(lot),2) " numbers
        are:": print#1: print#1
    160 rem assign the random values to the array
KI  165 for try=1 to max: for pik=1 to 6
CL  170 v%=rnd(0)*lot+1: if win%(v%,try) then 170
        : rem loop till un-used #
DH  175 win%(v%,try)=1: rem flag as used
MA  180 next pik, try
EB  185 rem ** got the numbers - build the strings **
AM  190 for pik=1 to max: for asn=1 to lot
MI  195 if win%(asn,pik) then out$(pik)=out$(pik)
        +right$(" [3 spaces] " +str$(asn),4)
AO  200 next asn, pik
HK  205 rem ** all ready - time to print **
DH  210 for spt=1 to 24 step 4: for prt=1 to max
EP  215 print#1,mid$(out$(prt),spt,4);
HD  220 next prt: print#1: next spt
MD  225 print#1: close 1: end
```

The Evil Swords Of Doom!

Beware as the evil sword slices through the screen and wipes any characters unfortunate enough to be in its way. Look out! Here comes another - you never know where the next one will strike. Before long, all characters have been slain by the EVIL SWORDS OF DOOM! Stay tuned until next issue for the conclusion of this exciting tale. (PHHH Gimme a break Chris - KH)

```
10 rem evil swords of doom
20 a$="M[g]M[g]M[g]M[g]MQQQQ[ | | | ][g]"
30 b$="[g] [g] [g] [g] "
40 print chr$(142)
50 print chr$(19)tab(rnd(1)*41)
60 fori=1to19:printa$;
70 rem delay here if desired
80 next i: print b$;: goto 50
```

Volume 6, Issue 05

C-64 Input Routine [159] **Dale Lambert,**
With Screen Editing! **Tupelo, MS**

This little INPUT substitute allows any characters to be entered and also allows full screen editing. It uses the input routine that BASIC uses in direct mode.

```
1 sys42336: for b=512 to 592: if peek(b)<>0 then next
2 in$="": poke peek(71)+256*peek(72)+1,0:
  poke peek(71)+256*peek(72)+2,2
3 poke peek(71)+256*peek(72),b-512: in$=mid$(in$,1)
```

Quick Screen Code [96, 137, 145, 163, 191] **Dale Lambert**
to ASCII Conversion

This line will convert screen code (in the variable S) to ASCII:

```
a=(s and 127)or((s and 64)*2)or((64-s and 32)*2)
```

C-64/VIC20 Mini-Datafier [203] **Dale Lambert**

This program will quickly and easily make DATA statements for a machine-language program.

All you have to do is put the starting address of the code in variable S, put the end address in E, put the starting line number of the DATA statements in Z, and the amount to increment the DATA line numbers by in variable I. For example:

```
1 s=49152(start): e=50000(end): z=1000(line #): i=10(incr)
```

And here's our program:

```
1 s=49152: e=49400: z=1000: i=10
2 print"■Sca■"z" data ";: if s>e then end
3 k=s+6: if k>e then k=e
4 for s=s to k: print mid$(str$(peek(s)),2)",";: next:
  print chr$(157);chr$(32) :rem 1 left, 1 spc
5 print"s=" s": e="e": z="z+i": i="i": goto2■s■";:
  poke631,13: poke632,13: poke198,2: end
```

Dale's Dazzler Try this on the 64:

```
1 a=192: b=200: c=53270: for i=1 to 1000 step .001:
  pokec,a: pokec,b: next
```

C64 Meets The Alien From The Cheap Sci-Fi Movie [81, 82, 91, 111, 146] **Giuseppe Amato**

Give this a listen, Earthlings:

```
1 s=54272: a=peek(162)and199: pokes+24,15: pokes+6,90:
  pokes+4,21: pokes+1,a
2 pokes+15,abs(99-a): goto1
```

VERIFIZER For Tape Users [1, 219] **Tom Potts, Rowley, MA**

The following modifications to the Verifizer loader (see the VERIFIZER page in this book) will allow VIC-20 and C-64 owners with Datasettes to use the verifizer directly (without the loader) and just SYS to activate it.

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.

Just run the program below, 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 special program 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 it has loaded, the screen will show FOUND VERIFIZER.SYS850. To activate VERIFIZER, enter SYS 850 (not the 828 as in the original program). To de-activate, use SYS 853. These moves in the SYS addresses were required because of the method used to store and retreive the program in the tape buffer.

If you are going to use your tape recorder to SAVE a program, you must turn off VERIFIZER first (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.

Make the following additions and changes to the present VERIFIZER loader listed on page 4 of any magazine, and at the beginning of the book.

```
     Line:
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, 247
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
```

Improved 1541 Head-Cleaning Program [181] **David Peterson, Irvine, CA**

Volume 6 Issue contained a program by Peter Boisvert which turned the 1541's motor on for 60 seconds to allow cleaning the head using a cleaning disk. This prompted David Peterson to write in with the following improvement. It turns the motor on, then steps the head slowly along the surface of the disk to utilize the entire cleaning surface. David Peterson explains how it works:

"After turning on the drive motor, the program peeks location $24 in drive RAM. This location contains the track number that the read/write head is currently at. After finding the head, the program steps it quickly to track 1, then slowly across the disk to track 35. Movement of the head is controlled by bits 0 and 1 of location $1C00 in drive RAM. After peeking $1C00, the head is moved outward to track one by cycling bits 0 and 1 of $1C00. To move the head outward the low bits are decremented (say 01 to 00 to 11 to 10 to 01 etc.). To move the head inward to track 35 the two low bits are cyclically incremented. The head is stepped twice for every track, since the stepper motor moves the head in 1/2 track steps. The NEW at the end of the program is not an attempt at program protection, it's there as drive protection. This direct method of stepping the head does not update location $24. If the program was immediately rerun, the drive head could end up being stepped to track 35 or to bump up against the stop at track 0. Therefore use the loop in line 280 to control how long the process takes."

Here's the new cleaning program; make sure you save it before running!

```
LM  100 rem* improved 1541 head cleaning prg *
LD  110 print" Sinsert cleaning disk and hit return"
NI  120 geta$: if a$<>chr$(13) then 120
AL  130 open 15,8,15: print#15,"m-e"chr$(126)chr$(249)
EE  140 rem locate head
HP  150 print#15,"m-r"chr$(24)chr$(0)
JD  160 get#15,a$: x=asc(a$+chr$(0))
EC  170 print" drive head at track #" x
IA  180 rem read $1c00
PH  190 print#15,"m-r"chr$(0)chr$(28)
EL  200 get#15,sc$: sc=asc(sc$+chr$(0))
FO  210 rem select bits 0 and 1
GN  220 bt=sc and 3
CD  230 rem # tracks to 1
ID  240 sp=2*(x-1)
MH  250 rem move head to track 1
BL  260 print" c stepping to track #1"
FJ  270 for y=1 to sp
EF  280 bt=bt-1: bt=bt and 3
KO  290 s=(sc and 252)or bt
IH  300 print#15,"m-w"chr$(0)chr$(28)chr$(1)chr$(s)
FE  310 next y
HL  320 rem step out to 35
IM  330 print"stepping out to track # 35. . ."
FB  340 print#15,"m-r"chr$(0)chr$(28)
CH  350 get#15,a$: sc=asc(a$+chr$(0))
CG  360 bt=sc and 3
CE  370 for y=1 to 68
GF  380 print" qq track #" int(y/2+1)
AF  390 print" QQQQ "
JM  400 bt=bt+1: bt=bt and 3
CG  410 s=(sc and 252) or bt
AP  420 print#15,"m-w"chr$(0)chr$(28)chr$(1)chr$(s)
FF  430 for d=1 to 220: nextd
HM  440 next y
FI  450 print#15,"m-e"chr$(232)chr$(249): close15
HF  460 new: rem to prevent re-running without a normal
        disk operation first
```

PRINT AT Update [188] **Stephen Gast, Champaigne. IL**

"In the Bits and Pieces column of Volume 6, Issue 3, a C64/VIC20 PRINT AT command was suggested:

```
poke 781,row: poke 782,col: sys 65520: print " message "
```

The above method utilizes the documented KERNAL routine PLOT. The general technique is a useful one but can be unreliable when accessed through the KERNAL jump table at 65520 ($FFF0).

If the carry flag is set, the routine will GET the current cursor position! Not exactly what we had in mind. To correct this the current row and column coordinates should still be placed directly into the register storage area in 781 ($030D9 and 782 ($030E). Then simply bypass the logic of the cursor get/set routine at $E50A and SYS directly to 58636 ($E50C). In both the VIC 20 and the Commodore 64 this will work:

```
poke 781,row: poke 782,col: sys58636: print " message "
```

Now let's talk a little about some other things you can do on a 64. First, the following line is an alternative to the above example:

```
poke 211,col: poke 214,row: sys 58640: print " message "
```

This enters the plot routine a little later and avoids two steps (big deal at ML speeds). But secondly, if you do this a lot in a program, here is a neat 25 byte machine language routine that makes life a little simpler:

```
0806 20 fd  ae jsr  $aefd  ;scan past the comma
0809 20 9e b7 jsr  $b79e ;put row in .X
080c 86 d6     stx  $d6    ;store row in TBLX (current cursor line #)
080e 20 f1  b7 jsr  $b7f1  ;scan past comma and put column in .X
0811 86 d3     stx  $d3    ;put column in PNTR (current cursor column #)
0813 4c 10 e5 jmp $e510 ;set the cursor
```

The following short BASIC program will place this routine in a REM statement:

```
10 rem xxxxxxxxxxxxxxxxxxxxxxxxx   [25 x's]
20 for x = 2054 to 2069: read y: poke x,y: next x
30 data 32,253,174,32,158,183,134,214,32,241,183,134,211,76,16,229
```

Be sure 25 x's follow the REM in line 10. After typing the program in, run it and delete lines 20 and 30. Save the remaining line 10 to disk as a program and simply load and use it as the first line of any program in which you want to easily be able to position the cursor. The syntax is now simplified to:

```
SYS 2054,row#,column#: PRINT " message "
```

Here are a couple of "C64 mode" peculiarities: The CAPS–lock key can be read in C64 mode, and the most interesting feature – 2Mhz. clock speed is available in C64 mode!

The CAPS–lock key can be checked with bit 6 in memory location $0001 of the C64 side. If bit 6 is set, then the CAPS–lock key is NOT pressed; if it is zero, the key is pressed. Example:

```
if (peek(1)and64) = 0 then print " caps lock on " .
```

Bit 0 in memory location $D030 controls the speed of the microprocessor. In C64 mode, this bit is normally zero, running the system at 1Mhz. If you set this bit, the computer will run at 2Mhz! Example:

```
poke 53296,peek(53296)or1
```

to set 2Mhz mode. The catch is that in C64 mode, the VIC video chip cannot operate at warp 2 and is disabled when 2Mhz mode is set, displaying a blank screen. However, for operations which do not need the video screen, such as assembling programs in machine language or sorting lists, the screen can be turned off for a speed increase of 100%!

For a simple demonstration, try the following program in C64 mode.

```
10 print " C64 at 2Mhz – Randy Linden "
20 print " Caps–lock down for 2Mhz, "
30 print " Caps–lock up for 1Mhz. "
40 for d = 0 to 36
50 read rl: poke 52992 + d,rl: ck = ck + rl: next d
60 if ck<>3571 then print " ?Data error! " : stop
70 sys 52992: print " Now installed. " : end
100 data 169,  11, 141,  20,   3, 169, 207, 141
110 data  21,   3,  96, 165,   1,  41,  64,  73
120 data  64,  10,  10,  42, 141,  37, 207, 173
130 data  48, 208,  41, 252,  13,  37, 207, 141
140 data  48, 208,  76,  49, 234
```

The code resides at $CF00–$CF25 in memory on the Commodore 64. When run, it changes the IRQ vector to point to a routine at $CF0B which scans the CAPS–lock key and turns on 1Mhz mode if it is up, or 2Mhz mode if it is down.

More B128 Bits From Liz Deal [101, 169 177]

1. COLLECT is a variant of the DOS native command VALIDATE (V). The B machine actually *thinks* at the time of validation. Example: if you've just written a file of 6 blocks but didn't close it, the directory shows 6 blocks and a *. At this point 'V' would get rid of the file, but COLLECT closes it. Not bad.

2. An important control byte exists at $258: Logical file number of CMD file. It's used for printing integers (line numbers) on a CMD device. It permits sending disk directories directly to a CMD device, so:

OPEN 4,4:CMD 4: CATALOG

Does just that to the printer. Neat.

3. Everybody knows that LIST is a harmless command. . .or is it? Try it on the B128 with a program from the plus 4 containing the keyword SCALE (that's token number $E9). The machine either crashes dead or ends up in the machine language monitor. And if that weren't enough, the program you just tried to list NEWs itself. Brilliant.

4. Locations $20–$21 are important in working dynamic strings. They hold temporary pointers to strings. Do not ever use them for anything else if you don't want your strings mangled, FRE crashing, and so on.

Un–Scratcher For Commodore Drives [162, 182]

Oops! Just scratch the wrong file by mistake and wipe out 3 hours of work? Don't panic – use the Un–Scratcher. If you haven't done any more saving since the scratch, your old file is still recoverable. The Un–Scratcher program below will ask you the type of drive you have (you may want to hard–code this into the program if you're always using the same drive), and display the filenames of scratched programs one by one, asking if you wish to un–scratch them. It will then write in the new directory information and validate the disk, asking for confirmation before each step. After that your life may continue normally since your precious work has been restored. Even if you have to enter it by hand, this is a routine that can really pay for itself!

```
CN | 100 rem save "0:un-scratch",8: rem ** rte/85
NL | 110 z$=chr$(0): cr$=chr$(13): sc=1: dr=0: rem dirsec
   |     + drvnum
FI | 120 print "** disk file un-scratcher **" cr$ "enter drive type"
LO | 130 print " a) 1541/2031" cr$ " b) 2040/4040" cr$ "
   |     c) 8050/8250"
FM | 140 input "your choice ";dt$: if dt$<"a" or dt$>"c" then 140
```

```
MP | 150 dt=18: bh=3: if dt$=>"b" then bh=17: if dt$="c"
   |     then dt=39
KE | 160 open 15,8,15: open 8,8,8,"#0"
EN | 170 print#15,"u1:"8;dr;dt;sc: rem read dir sector
GD | 180 print#15,"m-r"chr$(0)chr$(bh)chr$(2): get#15,nt$,ns$:
   |     rem next trk/sec
MG | 190 flag=1: for lk=0 to 7: ps=lk*32: print#15,"m-r"chr$(2
   |     +ps)chr$(bh)chr$(19)
KD | 200 get#15,sb$,ft$,fs$: if len(sb$) or len(ft$)=0 then 310
LE | 210 print "1st data track"asc(ft$)"sector"asc(fs$+z$)
BD | 220 print "filename ";: for name=1 to 16: get#15,n$: print n$;
   |     : next
FF | 230 print#15,"m-r"chr$(30+ps)chr$(bh)chr$(2): get#15,l$,h$
   |     : rem file size
LE | 240 print: print "size"asc(l$+z$)+256*asc(h$+z$)"block(s)"
KD | 250 input "un-scratch (y/n) ";us$: if us$<>"y" then 310
   |     : rem * nope
KN | 260 input "file type: s, p, u, r ";ft$: us=0
ML | 270 for chk=1 to 4: if ft$=mid$("spur",chk,1) then
   |     us=chk+128
OK | 280 next chk: if us=0 then 260: rem incorrect reply
KM | 290 print#15,"m-w"chr$(2+ps)chr$(bh)chr$(1)chr$(us)
CH | 300 print "done !!": print: flag=0
KE | 310 next lk
EB | 320 if flag then 350: rem no change in sector
AH | 330 input "write block to disk (y/n) ";yn$: if yn$<>"y" then 350
MK | 340 print#15,"u2:"8;dr;dt;sc
PA | 350 dt=asc(nt$+z$): sc=asc(ns$+z$): if dt then 170
   |     : rem more to go
NE | 360 if us=0 then 390: rem nothing to un-scratch
OG | 370 input "validate the diskette (y/n) ";yn$: if yn$<>"y"
   |     then 390
CO | 380 print#15,"v"+str$(dr): print ">> validating disk <<"
PF | 390 close8: close15: end
```

Hardware Device Number Change for the 2031 Drive [40]

The 2031 single drive can be hard-wired as device number 9 or 10. The number is determined by two diodes on the PC board, CR18 and CR19. Both diodes are normally connected for device number 8; snip one of the leads on CR18 for device 10 or CR19 for device 9.

C64 Doodle Screen [133, 222] **Tom White, Sudbury, ON**

If you've ever drawn pictures on the screen with the standard graphics and editor on the C–64, you've probably hit the RETURN key by accident more than once. This results in a READY or ?SYNTAX ERROR message partly wiping out your creation. To counter this menace, simply enter the following before you start your masterpiece:

```
poke 768,123: poke 769,164
```

While in this mode, all of the BASIC commands still work, so take care not to type LIST, RUN or any similar instruction that might ruin your picture. To return the error messages back to normal, type:

```
poke 768,139: poke 769,227
```

1541 Write–Protect Check [185] **Craig McQueen, Guelph, ON**

Have you ever wanted a routine to find out if there is a write–protect switch on a disk? All one has to do is read the value of $1C00, bit 4. If the bit is 0, then the write protect is on. Here is a memory–read routine to do the checking for you.

```
5 rem check for write-protect (1541)
10 open 15,8,15
20 print#15," m-r " chr$(0);chr$(28)
30 get#15,a$: a = asc(a$ + chr$(0))
40 if (a and 16) then 70
70 print " write protect is on! "
60 goto 80
50 print " no write protect "
80 close 15
```

C–64 Memory Fill ROM Routine [155] **Thomas Henry, North Mankato, MN**

Thomas writes:
"In the Volume 6, Issue 1 Bits & Pieces section, you described the use of the memory transfer subroutine contained within the BASIC ROM. One vital piece of information is missing, though. The memory transfer routine is not "intelligent". Specifically, it fails to work correctly if you attempt to move a block of data in the downward direction AND if the source block overlaps the destination block. In all other cases however, it works just fine.

By the way, the routine has this limitation since it was originally designed to spread apart BASIC lines in memory. The designers apparently never in-

tended for it to be used for any other purpose than making room for new BASIC lines in RAM. Of course, this type of memory transfer will always be occuring in an upward direction (from lower to higher adresses). Now that we know the limitation, however, we hackers can use it for other purposes as well.

Here's a neat trick that actually exploits the shortcomings just mentioned to good advantage. See if you can figure out how it works. (The following addresses are for the C–64; refer to the abovementioned issue for for the corresponding addresses in the PET/CBM and VIC–20.)

THE TASK: We wish to fill a block of RAM with a specific byte. We'll call this a memory fill subroutine. Possible uses include clearing a bit–map screen, setting colour memory to some value, filling a buffer with zeroes, etc.

THE SOLUTION: Suppose the addresses of the start of the block to be filled is START and the last address of the block is END. For example sake, let's imagine we wish to fill this block with zeros. Perform the following steps:

1. Store a zero (or whatever byte you wish to fill memory with) in location END
2. Store address END in location $58 (low byte, high byte).
3. Store address END + 1 in location $5A.
4. Store address START + 1 in location $5F
5. Call subroutine $A3BF (The memory transfer subroutine)

The block from START to END (regardless of length or location) will be filled with the specified byte – zeros in this case.

One limitation of the block fill routine is that it cannot be accessed from BASIC. Apparently the POKE number or SYS number evaluator plays havoc with locations $58 through $60. However, the routine works just great with machine language and is far simpler to use than "rolling your own".

Relocate! [19]

When creating sprite files, high–res screens, character sets and the like, you don't always know where in memory you'll want to put them. You'd also like those files to be LOADable programs that will go into whatever spot you want. For example, some drawing packages create a high–res screen PRG file on disk that can be LOADed into memory starting at $2000 (8192). If you wish to use that picture but put it at, let's say, $E000, you need to change the picture file's load address on disk. Guess what RELOCATE does? Just tell it your drive type, the filename of the program, and your desired load address, and it changes the load address of the file. Right now. Yes, it *is* fast, because it goes directly into disk memory to change the first two bytes of the file and writes the block back to the disk surface.

```
HD  100 rem save " 0:relocate " ,8
AF  105 rem ** rte/85 - allows quick change of prg load address **
CO  110 :
GK  115 z$ = chr$(0)
CK  120 print " drive type: "
GH  125 input " 1)1541/2031, 2)4040, 3)8050/8250 " ;d: if d<1 or d>3
        then 125
DD  130 if d = 1 then dl = 144: dh = 2: di = 4: dt = 18: bl = 0: bh = 3
        : rem 1541/2031
HC  135 if d = 2 then dl = 150: dh = 67: di = 4: dt = 18: bl = 0: bh = 17
        : rem 2040/4040
OF  140 if d = 3 then dl = 96: dh = 67: di = 8: dt = 39: bl = 0: bh = 17
        : rem 8050/8250
FA  145 :
AJ  150 input " drive #, filename " ;dr,f$: f$ = str$(dr) + " : " + f$
HE  155 input " new start address (decimal) " ;sa : sh% = sa/256:
        sl = sa-256*sh%
MP  160 open 15,8,15: open 8,8,8,(f$): get#8,a$: if st then close8:
        stop
FL  165 print#15, " m-r " chr$(dl)chr$(dh): get#15,s$: rem sector
IP  170 print#15, " m-r " chr$(dl + di)chr$(dh): get#15,i$: rem index
ID  175 s = asc(s$ + z$): i = asc(i$ + z$) + 1
II  180 close 8: open 8,8,8, " #0 "
AH  185 print#15, " u1: " 8;dr;dt;s: rem read in directory track/sector
BD  190 print#15, " m-r " chr$(bl + i)chr$(bh)chr$(2) : rem get 1st data
        block ptr
OM  195 get#15,t$,s$: t = asc(t$ + z$): s = asc(s$ + z$)
HF  200 print#15, " u1: " 8;dr;t;s: rem get first data block
DM  205 print#15, " m-w " chr$(bl + 2)chr$(bh)chr$(2) chr$(sl)
        chr$(sh%)
BP  210 print#15, " u2: " 8;dr;t;s: rem write block back
IE  215 close 8: close 15
GE  220 print " ** address changed ** "
BO  225 end
```

SAVERIFY [186] **Bob Hayes, Winnipeg, Manitoba**

This is a short program which enables the 64 (and possibly other Commodore?) owner to SAVE and VERIFY a program with one command. The format is:

SYS(address)" filename",8

Where the address is the start of the machine language code (relocatable in the BASIC loader).

Here is the assembly:

```
start  jsr  $e1d4
       jsr  $e159
       lda  #$01
       sta  $93
       bit  $00a9
       sta  $0a
       jsr  $e16f
       rts
```

And the BASIC loader:

```
10 rem ** saverify -- bob hayes **
20 rem ** wpg, man. canada     **
25 sa=828: rem start address-note: relocatable
30 q$=chr$(34): a=0: for x=0 to 18: read q: poke sa+x,q:
   a=a+q: next
40 print: print" format: sys" sa;q$" filename" q$",8"
50 end
60 data  32, 212, 225,  32,  89, 225, 169,   1
70 data 133, 147,  44, 169,   0, 133,  10,  32
80 data 111, 225,  96
```

Double Verifizer [1, 208] **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. The program below, 'DOUBLE VERIFIZER' solves that problem by showing the two-letter verifizer code on both the first and second row of the TV screen. The program uses the interrupt vector to update the screen every 1/60 of a second.

To use Double Verifizer, 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
```

Corrupting RAMTAS Update [195] **Yijun Ding, Pittsburgh, PA**

"Corrupting RAMTAS Routine" in Bits and Pieces Volume 6, issue 4 mentioned the fact that $A000 will contain $55 after a reset. But there is more. RAM from $FD30–$FD4F will be written with the contents of the corresponding ROM, as the routine at $FD15 ($FF8A, reset vectors in $0314–$0333) is called in a reset process. Actually, the RAM at $FD30–$FD4F will be corrupted every time $FD15 is called.

Finding the Missing File **Jeffery Coons, Lake Ridge, Virginia**

If a program bombs because it needed some file that wasn't on the disk, you can find out what file the program wanted with this one–liner:

```
for i=0 to peek(183)-1: poke 1024+i,peek(peek(188)*256
                        +peek(187)+i): next
```

The name of the last file used will be displayed at the top left–hand corner of the screen. You have to POKE to the screen in this manner because PRINTing will corrupt the last character in the string. Users with ROM version 2 will have to also POKE to colour memory (at 55296+i) or make sure there is some text already on the top line of the screen.

LOAD & RUN Trick [21, 46, 182] **Chris Wong**

. . .A really neat load and run trick: After you type

```
load"filename",8,1
or load"filename",8:
```

Press shifted RUN/STOP instead of RETURN. The program will automatically RUN itself after loading. It eliminates the old load/return/run/return routine, easing up loading a bit.

Check For Device Present [41] **Dave Pollack, E. Brunswick ,NJ**

As most every C-64 user knows, the 'DEVICE NOT PRESENT" message and consequent crash is not the most pleasant experience in the world to endure. Believe me, I've been searching for close to a year for ANY solution that will work. It was not that obvious. I stumbled upon it quite by accident after coding a small routine that provided a way for me to print the value of the 'ST' variable after multiple I/O operations. If you do that you'll notice something interesting. An OPEN followed by an immediate CLOSE will not hang the computer even if the device is not present, but it allows you to interrogate ST which returns a nonzero result in this case.

If you use the following code, your program will be able to check for DEVICE NOT PRESENT and continue without bombing.

```
100 open 15,8,15: close 15
110 if st<>-128 then 160
120   print "!! DRIVE NOT PRESENT !!"
130   print "## check drive power and cables, then press a key ##"
140   get a$: if a$= "" then 140: rem wait for a key
150   goto 100
160 rem program continues. . .
```

Word-Wrap For VIC, 64, PET, etc. [146] **Gary Royal, Chicago, IL**

There never seems to be enough columns on the screen to display what you want to print on it. And there's nothing uglier than a word hanging partly on the end of one line and at the beginning of another. Whining about it does no good (I've tried), but word-wrap does. Place the string you want wrapped in 'w$', the desired line width in 'w', and call this routine.

```
100 rem* recursive word-wrap routine *
110 rem* put string in w$,
120 rem* line width in w
130 :
140 if len(w$)>w then 160
150 print w$: return
160 p=0: for i=w to 1 step-1
170 if p=0 and mid$(w$,i,1)="  " then p=i
180 next: h$=right$(w$,len(w$)-p)
190 w$=left$(w$,p): gosub150
200 w$=h$: goto 140
```

Since strings in Microsoft BASIC can be up to 255 characters long, you can easily squeeze five screen lines into w$ with the peace of mind that can only

come from the knowledge that it will be formatted legibly. But beware! the routine is recursive, and assumes that words in the text will be separated by spaces. If the length of w$ is greater than 'w' and 'w$' contains no spaces it will loop forever, so avoid hyphenated words that might be longer than your desired line length (or modify line 170 to look for hyphens, too).

Visible "searching" Messages [127, 127, 215]

Terry Montgomery, Auckland, New Zealand

In direct mode you get 'SEARCHING' and 'FOUND' messages that tell you what is going into the computer. These messages can be extremely helpful, especially when using tape. But when LOAD statements are encountered in program mode, the messages are suppressed. During program development, it would be nice to see what's going on a bit more. Here are two ways to see these messages from a running program:

1) Use GOTO instead of RUN to start the program. If the first line is 0, GOTO doesn't need a line number specified.

2) POKE 157,128 to flag direct mode. This can be turned off by POKE 157,0. This way you can get messages from one part of the program and block them from others.

C–64 Scroll Down Routine [85, 155]

Chris Johnson, Toronto, Ont.

In Volume 5, Issue 2 of The Transactor, Paul Blair reported a ROM routine that scrolled down the screen of a C–64. He also mentioned that it "left some pointers a bit untidy . . . a PRINT or two seems to restore order".

I found that a PRINT or two did *not* set things right; however, resetting the screen line link table did. The following routine clears the link table before and after calling the scroll–down routine.

The syntax to use is:

SYS address, n, topline

Where 'n' is the number of times you want the screen to be scrolled down one line and topline (0 to 24) is the last line not to be scrolled. All the lines below this will be scrolled down x times.

To change the location of the routine, just change the value of s in line 110. The loader will make the necessary changes to the machine code.

```
AF  100 rem* c-64 scroll down *
MO  110 s=49152: rem start address (relocatable)
OL  120 for i=s to s+33: read a: poke i,a: next
PH  130 print"** scroll down - syntax: "
DI  140 print"sys"s",n,topline"
LA  150 print" Where 'n' = number of lines to scroll"
EB  160 :
OI  170 if s=49152 then end
GM  180 u=s+22: ju=s+7: r=s+34: jr=s+4
CB  190 jj=s+18
AG  200 poke ju+1,u/256: poke ju,u-256*peek(ju+1)
PN  210 poke jj+1,r/256: poke jj,r-256*peek(jj+1)
BE  220 poke jr+1,r/256: poke jr,r-256*peek(jr+1)
KF  230 :
HB  240 data  32, 241, 183, 142,  34, 192,  32,  22
KD  250 data 192,  32, 241, 183, 134, 214,  32, 101
FH  260 data 233, 206,  34, 192, 208, 248, 162,  24
AD  270 data 181, 217,   9, 128, 149, 217, 202, 208
GJ  280 data 247,  96
```

Easy 'RESTORE X' Using TransBASIC

[48, 72, 110]

Andy Hochheimer, Wallaceburg, Ont

I have been using a lot of DATA statements in programming for quite a while. 99% of the time I have to RESTORE then search for my data on a specific line number before reading again. In Transactor Volume 5 Issue 3 was this 'RESTORE X' program from Gary Kiziak, which allowed a RESTORE to a specific line number:

```
10 restr=828: for k=restr to restr+31:read j:poke k,j: next
20 data  32, 253, 174,  32, 158, 173,  32, 247
30 data 183,  32,  19, 166, 175,   5, 162,  17
40 data  76,  55, 164, 165,  95, 233,   1, 133
50 data  65, 165,  96, 233,   0, 133,  66,  96
60 rem format: sys restr x
```

I've found a shorter and easier way to RESTORE X, using TransBASIC:

10 doke 65,line(x)+4

This incredible program line *does* work; location 65 is the Current DATA Address. It restores the pointer to the first byte of line X. The 4 is added to avoid reading the last data element of the previous line. This is a small sample of the great things you can do with TransBASIC!

Sneaky Saves [149, 182] **Terry Pridham, Belmont, Ont.**

In Vol 5 issue 3, "Unveiling The Pirate Part 2: Programming Sleight of Hand" – 'Ye Olde Standbye', where by using a shifted–space before the filename within quotes produces a directory that shows two quotes followed by the filename:

save " 0:[Shift-space]filename " ,8

In the directory it becomes:

3 " " filename prg

By experimenting with it, I found even more ways to twist the minds of Pirates (as if they weren't in the first place). Ever see directories where the name of the program is in reverse field? Well here's how it's done. Type:

save,1 quote, drive number, colon, 1 quote, rvs on, 1 delete, 2 inserts, shift–M, rvs on, rvs off, filename, quote, comma, device number

When done, it will appear something like:

save " 0: Mr filename " ,8

In the directory, it will show the block count and the first quote where it would normally appear. The shifted M causes a carriage return (because a shifted reverse M is a '13') and the filename will appear right under the block count in reverse field. The file type indicator (i.e. "prg") and the spaces preceding it will also appear in reverse field.

Try adding a couple DELs, or even cursor control characters, by hitting 1 Insert for every control character you wish to include immediately before the filename. However, you must remember what characters are in this "prefix" in order to LOAD that file. Experiment and have fun!

Sanitation Engineer [129, 130, 132] **Fred Simon**
Gibbsboro, New Jersey

Did you ever have to wait for several minutes while your computer collected "garbage" strings? Garbage collection on the C–64 has been known to take more than twenty minutes when a large number of strings need be processed. With the program "Sanitation Engineer", active strings are collected lightning fast.

What Is Garbage Collection?

Each time the Basic interpreter encounters a new string variable definition, it builds that string character by character in high memory, working downward from location 40960. If a string variable is changed, the old string remains in memory as "garbage". If the available free memory is less than the maximum length of a couple of strings, or if the Basic command FRE(0) is issued, the garbage collection routine is called. This routine looks at each string variable to find the one stored highest in memory, moves all of the other strings down by the length of this string, and then copies the string to the top of available memory. The length of time it takes to complete this task depends only on the number of strings and not their length.

To see garbage collection at work, try this program:

```
10 d=500: dim x$(d)
20 for j=0 to d: x$(j)=str$(j): next
30 print "starting collection. . ."
40 t=ti: j=fre(0)
50 print (ti-t)/60 " seconds"
```

Change the value of D in line 10 to see the effect of increasing the number of strings.

Faster Collection

One way to speed up garbage collection is to first copy the string memory to a buffer area (Sanitation Engineer uses the area located underneath the Kernal ROM). Each active string can then be pulled out of the buffer and written to the clean string area. The bottom of the string memory is then the bottom of the last active string copied from the buffer. Sanitation Engineer is written as a "patch" to the Basic operating system. It uses the area of memory from 51740–52223 for the garbage collection routines. Thus, it can be used with the DOS Wedge and leaves 49152–51739 free for other machine language routines.

Type in and Save Sanitation Engineer. A mistake in one of the Data statements could cause your computer to lock-up when the routine is executed. A checksum is included to reduce the chance of errors. When you Run the program, Basic ROM is first copied to RAM. The new address for the Sanitation Engineer is written over the old collection routine. In addition, the READY. prompt is changed to READY! to remind you that Basic has been modified. If you hit Run/Stop-Restore, the Sanitation Engineer will be deactivated. To reactivate, just type SYS 51740. Try the test program you typed in earlier. Change D to 5000 and try again. No more delays!!

```
PI 10 rem save"0:sanitation 64",8
DD 100 rem sanitation engineer
FN 110 rem for the commodore 64
BP 120 rem by fred simon 8/85
HO 130 ck=0: for i=51740 to 52223: read d
BP 140 poke i,d: ck=ck+d: next
EG 150 if ck=63591 then sys51740: end
GK 160 print "error in data statements": stop
OB 170 :
MD 180 data 120, 169,  55, 133,   1, 169, 160, 133
HL 190 data   3, 160,   0, 132,   2, 177,   2, 145
AN 200 data   2, 136, 208, 249, 230,   3, 165,   3
JH 210 data 201, 192, 208, 241, 169,  54, 133,   1
NC 220 data  88, 169,   5, 141, 143, 183, 169,  33
LG 230 data 141, 125, 163, 162,   2, 189,  83, 202
BF 240 data 157,  38, 181, 202,  16, 247,  96,  76
JK 250 data  86, 202, 169,   0, 141, 239, 203, 169
EG 260 data  15, 133, 250, 169, 224, 133, 249, 165
NI 270 data  52, 141, 240, 203,  56, 229,  50, 201
NE 280 data  19, 144,  22, 233,   3, 133, 250, 165
ND 290 data  50, 105,   0, 133, 249, 165,  56, 229
DB 300 data  52, 105,   1, 197, 250, 176,   2, 133
EK 310 data 250, 165,  56, 141, 242, 203, 165,  55
GD 320 data 141, 241, 203, 133,  51,  24, 240,   1
GI 330 data  56, 173, 242, 203, 133,  52, 233,   0
KJ 340 data 133, 251, 105,   0, 133, 252, 165,  50
KB 350 data 105,   1, 133, 254, 165,  45, 233,   6
HJ 360 data 133,  95, 165,  46, 233,   0, 133,  96
NK 370 data 165,  47, 133, 253, 165, 251, 205, 240
BL 380 data 203, 144,  51, 229, 250, 133, 248, 165
EA 390 data  52, 229, 251, 229, 248,  73, 255, 105
OK 400 data   2, 197, 248, 144,   2, 165, 248, 205
JI 410 data 240, 203, 176,   5, 173, 240, 203, 233
DI 420 data   0, 133, 251,  32, 138, 203, 166,  48
BO 430 data  32, 243, 202, 176,   9,  32,  39, 203
CC 440 data 165, 251, 133, 252, 144, 182,  96,  24
GC 450 data 165,  95, 105,   7, 133,  95, 144,   2
KF 460 data 230,  96,  69,  47, 208,   4, 228,  96
PM 470 data 240,  31, 160,   0, 177,  95, 200,  81
FC 480 data  95,  16, 228, 177,  95,  16, 224, 160
DG 490 data   4, 177,  95, 197, 251, 144, 217, 197
JA 500 data 252, 176, 212,  32, 170, 203, 144, 208
NF 510 data  96,  24,  96,  32,  83, 203, 176, 249
MA 520 data 160,   2, 177,  95, 197, 251, 144,  10
```

```
NJ  530 data 197, 252, 176,   6,  32, 170, 203, 144
CI  540 data   2,  96,  24, 169,   3, 101,  95, 133
DJ  550 data  95, 144,   2, 230,  96, 197, 253, 208
KJ  560 data 223, 228,  96, 208, 219, 240, 212,  24
CL  570 data 165, 253, 133,  95, 134,  96,  69,  49
JE  580 data 208,   4, 228,  50, 240,  39, 160,   2
LH  590 data 177,  95, 101,  95, 133, 253, 200, 177
AH  600 data  95, 101,  96, 170, 160,   0, 177,  95
KG  610 data 200,  81,  95,  16, 218, 160,   4, 177
HL  620 data  95,  10, 105,   5, 101,  95, 133,  95
MJ  630 data 144,   3, 230,  96,  24,  96, 165, 248
EA  640 data 133,  79, 165, 249, 133,  89, 160,   0
OM  650 data 132,  78, 132,  88, 166, 250, 232, 177
EE  660 data  78, 145,  88, 200, 208, 249, 230,  89
IF  670 data 230,  79, 202, 208, 242,  96,  72, 120
JD  680 data 169,  53, 133,   1, 104, 197, 248, 144
KM  690 data   5, 229, 248,  24, 101, 249, 133,  79
DI  700 data 136, 177,  95, 133,  78, 136,  56, 165
CO  710 data  51, 241,  95, 133,  51, 200, 145,  95
PO  720 data 165,  52, 233,   0, 133,  52, 200, 145
HG  730 data  95, 136, 136, 177,  95, 240,   9, 168
OD  740 data 136, 177,  78, 145,  51, 152, 208, 248
ND  750 data 169,  54, 133,   1,  88,  24, 165, 254
IH  760 data 229,  52,  96,   0,   0,   0,   0,  67
JG  770 data  49,  57,  56,  53,  32,  70,  46,  83
DL  780 data  73,  77,  79,  78
```

Some C128 Bits [174, 190] **Perry Shultz, Miami, Florida**

Ornament and Happy New Year In High-Res

```
9 graphic1: scnclr: color1,5: for u = 1 to 50 step3: circle1,160,75,u,60-u:
  next: color1,2: for r = 9 to 85 step5: circle1,160,r/9,r*2,r*3,,,,72: nextr:
  char1,13,18," happy new year ",1
```

Notes: The line number must be 9 or less. Type line with no spaces. After entering the last character, cursor back anywhere in the line then return.

Multiple Circle, Triangle, and Square High-Res Draw Routine

```
5 graphic1,1: for i = 25 to 300 step9: circle1,i,100,20,18,,,,120: next:
  for i = 25 to 300 step9: circle1,i,20,20, 18,,,,45: next:
  for i = 25 to 300 step9: circle1,i,175,20,,,,,90: next
```

```
10 graphic1: scnclr: for r=3 to 100 step6:
   circle1,160,130,r+20,r+18,,,,120: nextr
15 graphic1: scnclr: for r=3 to 100 step4:
   circle1,r+100,130,r+20,r+18,,,,120: nextr
20 graphic1,0: scnclr: for r=3 to 100 step4:
   circle1,160,110,r+20,r+18,,,,100: nextr
25 graphic1,0: scnclr: for r=3 to 100 step4:
   circle1,99+r,110,r+20,r+18,,,,100: nextr
30 graphic1,0: scnclr: for r=3 to 100 step4:
   circle1,160,110,r+20,r+18,,,,90: nextr
35 graphic1,0: scnclr: for r=3 to 120 step3:
   circle1,r+ 70,r+20,r+20,r+18,,,,90: nextr
40 graphic1,0: scnclr: for r=3 to 100 step4:
   circle1,160,110,r+20,r+18,,,,150: nextr
45 graphic1,0: scnclr: for r=3 to 120 step3:
   circle1,r+75,99,r+20,r+18,,,,30: nextr
50 graphic1,0: scnclr: for r=3 to 120 step3:
   circle1,r+100,95,100,r+10,,,,75: nextr
55 graphic1,0: scnclr: for r=7 to 100 step2:
   circle1,160,r+60,r+55,r+3,,,,72: nextr
```

More Ideas

Redefine two function keys as graphic 0 (textscreen), graphic 1 (hi–res screen) — this enables screen change with one keytouch.

With the 160 bytes per line, I hope to see many new exciting 1 liners.

Some Amiga Bits and Pieces

Notes About CLI

CLI, Amiga's Command Line Interface, is your interface to AmigaDOS. You can access CLI by clicking its icon on your WorkBench disk – the CLI icon appears if the "CLI on" option is chosen in "Preferences". When a DOS command is entered, the system looks for the command in the current directory, and if not found, in the subdirectory C on the SYS: disk (the disk that was booted with). See the article in this issue for a brief description of the DOS commands.

The disk–oriented nature of the DOS commands makes for a flexible system, since you can add and change commands at will. With a single drive though, it

can be a problem doing operations with a disk other than SYS: (the one in the drive). For example, if you wish to get a directory of another disk, you can't just switch disks and type DIR because the system will ask for the SYS: disk again (by volume name) and then do a DIR, giving you the directory of your original disk. Since AmigaDOS is a fairly flexible and powerful system, there are many ways of getting around the problem; here are a few suggestions:

1) The standard method is to refer to the new disk by name when giving the DOS command, for example to get a directory of a disk called "Utilities", you could just enter:

```
dir utilities:
```

The system would then put up a requester asking you to insert volume "utilities" in the drive, and would give you a directory after you had done so. You can work with any file or directory on the new disk in this way, for example:

```
type utilities:stuff/TextFile
```

. . .would display the file "TextFile" in the sub-directory "stuff" on the disk "utilities". This method works fine when you know the volume name of the disk you're interested in (which you should, since you've thoughtfully written it on the disk label, right?), and you only want to use the disk a few times and don't mind swapping disks back and forth.

2) If you wish to switch to a new disk for awhile to perform several commands, and the new disk has those commands on it (usually in the C sub-directory), you can just change the assignment of C: , telling the system to look elsewhere for commands. For example, if from the original disk you typed:

```
assign c: utilities:c
```

You would be prompted to insert volume "utilities:", and the C sub-directory on that disk would then be searched for all DOS commands subsequently issued.

Likewise, you could re-assign the current directory using the CD command, as in:

```
cd utilities:c
```

The disadvantage with this approach is that it locks you into C as the current directory.

3) A more direct approach for using a new disk which also contains the DOS commands is to refer to the disk explicitly when issuing the command, preventing DOS from requesting the SYS: disk. For example, if you wanted a directory of any old disk laying around (remember, it MUST contain the

required DOS command – in this case DIR – in the C directory), just pop in the new disk and type:

```
df0:c/dir
```

That way you are referring to Drive 0 (not a specific volume), C directory, then finally the command name. This is a handy technique for little one-time commands such as a DIR or TYPE when you don't feel like typing in or don't know the new disk's volume name.

4) A favourite trick used by many is COPYing all or some of the DOS commands into RAM and then assigning C: to RAM to tell the system to look there for the commands. You could use the following sequence of commands, possibly in your startup-sequence batch file, to accomplish this:

```
makedir ram:c  ;make c sub-directory in RAM:
copy c: ram:c  ;copy entire c sub-directory to RAM:
assign c: ram: ;assign ram as new source of commands
```

This seems to be the ultimate solution at first glance, since all of your commands execute out of RAM at lightning speed, and you're never bound to a disk when issuing a command. The disadvantage (there had to be one) is that you use up lots of RAM, and also (OK, two) it takes a long time to copy all of those commands. Nonetheless, some people have enough RAM and enough time that this really **is** the ultimate solution to fast and flexible DOS commands.

5) A variation on the above RAM technique is my favorite, thought up by Amiga-buff Rico Mariani. Pick your most-used DOS commands, for example DIR, LIST, COPY, ASSIGN, CD, and TYPE, and copy them to RAM. Then assign names to each of those files, and use those new names in lieu of the command names. (ASSIGN is just a way of setting up a new name to refer to a volume, directory, or file.) As a confusion-avoiding convention, make the assigned names identical to the command names, except for the required colon (:) at the end. The example below should clear up any confusion (you could use this in your startup-sequence).

```
copy :c/dir to ram:
copy :c/copy to ram:
copy :c/cd to ram:
copy :c/type to ram:
assign dir: ram:dir
assign copy: ram:copy
assign cd: ram:cd
assign type: ram:type
```

Now, with those assignments in place, when you wish to do a DIR, just type dir: (with the colon at the end). This will get the dir command from RAM,

executing it quickly, and you don't have to have the dir command on the disk currently in the drive. Also, you haven't use up tons of RAM, since you've only copied the commands you need. Obviously the assignments aren't needed at all, since you could just use "ram:dir" for the same effect, but the assignments make things just a bit clearer and easier to type. Incidentally, you can use assign whenever you'd like to use an alias to refer to a directory or file. Tired of typing "execute" all the time? Just do an:

```
assign !: c/execute
```

and use "!:" instead of the word "execute" at any time. Assigns are system-wide, not just for the current window, so your assignments will last until re-boot (and beyond, if you put them in the startup-sequence).

Index