```
                   ########
             ##################
         ######            ######
      #####
    #####  ####  ####      ##      #####   ####  ####  ####  ####  ####   #####
  #####    ##    ##      ####    ##   ##   ##  ###     ##    ####  ##   ##   ##
 #####    ########     ##  ##   ##         #####           ##    ## ## ##   ##
#####    ##    ##    ########  ##   ##   ##  ###     ##    ##  ####   ##   ##
#####  ####  ####  ####  ####  #####   ####  ####  ####  ####  ####   ######
#####                                                                    ##
 ######            ######           Issue #6
   ##################            Sept. 5, 1993
       ########
```

Editor's Notes:
by Craig Taylor

School, 2 jobs, a play, and other work has seriously restricted the amount of time that I am able to devote to Commodore Hacking. What I am looking at doing now is to stop writing articles for Commodore Hacking and hoping that I'll still have enough time to write these notes, and organize and pull all the other articles by other contributors together. The two articles I had hoped to include in this issue : the one about multi-tasking and about the 1351 have again been delayed. I've decided to go ahead and release issue 6 and hopefully will have them in the next issue. (Remember: You get what you pay for.. *smile*)

As always, Commodore Hacking is constantly looking for articles, notes, short little programs, what-not on any side of the Commodore 64 and 128 - primarily on the technical or hardware side. If you think you have something in mind, or already written then feel free to drop me a line letting me know.

In regards to several queries recently about reprinting individual articles that have appeared in Commodore Hacking. You may reprint Commodore Hacking and redistribute in whole, freely. For use of individual articles you _must_ contanct the individual author as they still retain rights to it. Please see the legal notice / mumbo below.

I recently recieved some mail from wolfgang@halcyon regarding a disk magazine that he was in the process of starting and he has written the following preview of a disk magazine that looks to be exciting:

> "_Scenery_, a new disk-magazine focusing on the american and european demo scenes, will soon be available for download at a site/BBS near you! With articles on everything from coding to instrument synthesis to art, _Scenery_ will be the definitive word when it comes to creating a demo, or simply coding in general. Articles are being written by some of the top names in the scene, and promise to help everybody from the Basic Baby to the ML Mogul! Set to be released mid-August, _Scenery_ will hopefully be a worthy edition to the likes of Coder's World and C=Hacking. I am making the magazine available on various Internet sites, so look for it. We are always on the lookout for art, music, and coding talent, and if you'd be interested in submitting an article for publication, or simply have a question or comment, please mail me at 'wolfgang@halcyon.com'. Thanks.. and see you on the Net!"

==

Please note that this issue and prior ones are available via anonymous FTP from ccosun.caltech.edu under pub/rknop/hacking.mag and via a mailserver which documentation can be obtained by sending mail to "duck@pembvax1.pembroke.edu" with a subject line of "mailserver" and the line "help" in the body of the message.

==

NOTICE: Permission is granted to re-distrubte this "net-magazine", in whole, freely for non-profit use. However, please contact individual authors for permission to publish or re-distribute articles seperately. A charge of no greather than 5 US dollars or equivlent may be charged for library service / diskette costs for this "net-magazine".

==

In This Issue:

DYCP - is a name for a horizontal scroller, where characters go smoothly up and down during their voyage from right to left. One possibility is a scroll with 8 characters - one character per sprite, but a real demo coder won't be satisfied with that.

Opening the borders

VIC has many features and transparent borders are one of them. You can not make characters appear in the border, but sprites are displayed in the border too.

A Heavy Duty Power supply for the C-64

This article describes how to build a heavier duty power supply for your Commodore 64 computer and includes a full schematic in GeoPaint format.

LZW Compression

LZW is perhaps the most widely used form of data compression today. It is simple to implement and achieves very decent compression at a fairly quick pace. LZW is used in PKZIP (shrink),PKARC (crunch), gifs,V.42bis and unix's compress. This article will attempt to explain how the compression works with a short example and 6502 source code in Buddy format.

THREE-KEY ROLLOVER for the C-128 and C-64.

This article examines how a three-key rollover mechanism works for the keyboards of the C=128 and C=64 and will present Kernal-wedge implementations for both machines. Webster's doesn't seem to know, so I'll tell you that this means that the machine will act sensibly if you are holding down one key and then press another without releasing the first. This will be useful to fast touch typers.

The Demo Corner: DYCP - Horizontal Scrolling

by Pasi 'Albert' Ojala (po87553@cs.tut.fi or albert@cc.tut.fi))
Written: 16-May-91 Translation 02-Jun-92

DYCP - too many sprites !?

DYCP - Different Y Character Position - is a name for a horizontal scroller, where characters go smoothly up and down during their voyage from right to left. One possibility is a scroll with 8 characters - one character in each sprite, but a real demo coder won't be satisfied with that.

Demo coders thought that it looks good to make the scrolling text change its vertical position in the same time it proceeded from the right side of the screen to the left. The only problem is that there is only eight sprites and that is not even nearly enough to satisfy the requirements needed for great look. So the only way is to use screen and somehow plot the text in graphics, because character columns can not be scrolled individually. Plotting the characters take absolutely too much time, because you have to handle each byte seperately and the graphics bitmap must be cleared too.

Character hack

The whole DYCP started using character graphics. You plot six character rows where the character (screen) codes increase to the right and down. This area is then used like a small bitmap screen. Each of the text chars are displayed one byte at a time on each six rows high character columns. This 240 character positions big piece of screen can be moved horizontally using the x-scroll register (three lowest bits in $D016) and after eight pixels you move the text itself, like in any scroll. The screen is of course reduced to 38 columns wide to hide the jittering on the sides.

A good coder may also change the character sets during the display and even double the size of the scroll, but because the raster time happens to go to waste using this technique anyway, that is not very feasible. There are also other difficulties in this approach, the biggest is the time needed to clear the display.

Save characters - and time

But why should we move an eight-byte-high character image in a 48-line-high area, when 16 is really enough ? We can use two characters for the graphics bitmap and then move this in eight pixel steps up and down. The lowest three bits of the y-position then gives us the offset where the data must be plotted inside this graphical region. The two character codes are usually selected to be consecutive ones so that the image data has also 16 consecutive bytes. [See picture 1.]

Demo program might clear things up

The demo program is coded using the latter algorithm. The program first copies the Character ROM to ram, because it is faster to use it from there. You can easily change the program to use your own character set instead, if you like. The sinus data for the vertical movement is created of a 1/4 of a cycle by mirroring it both horizontally and vertically.

Two most time critical parts are clearing the character set and plotting the new one. Neither of these may happen when VIC is drawing the area where the scroll is, so there is a slight hurry. Using double buffering technique we could overcome this limitation, but this is just an example program. For speed there is CLC only when it is absolutely needed.

The NTSC version is a bit crippled, it only covers 32 columns and thus the characters seem to appear from thin air. Anyway, the idea should become clear.

Want to go to the border ?

Some coders are always trying to get all effects ever done using the C64 go to the border, and even successfully. The easiest way is to use only a region of 21 pixels high - sprites - and move the text exactly like in characters. In fact only the different addressing causes the differences in the code.

Eight horizontally expanded sprites will be just enough to fill the side borders. You can also mix these techiques, but then you have the usual "chars-in-the-screen-while-border-opened"-problems (however, they are solvable). Unfortunately sprite-dycp is even more slower than char-dycp.

More movement vertically

You might think that using the sprites will restrict the sinus to only 14 pixels. Not really, the only restriction is that the vertical position difference between three consequent text character must be less than 14 pixel lines. Each sprites' Y-coordinate will be the minimum of the three characters residing in that sprite. Line offsets inside the sprites are then obtained by subtracting the sprite y-coordinate from the character y-coordinate. Maybe a little hard to follow, but maybe a picture will clear the situation. [See picture 2.]

Scrolling horizontally is easy. You just have to move sprites like you would use the character horizontal scroll register and after eight pixels you reset the sprite positions and scroll the text one position in memory. And of course, you fetch a new character for the scroll. When we have different and changing sprite y-coordinates, opening the side borders become a great deal more difficult. However, in this case there is at least two different ways to do it.

Stretch the sprites

The easiest way is to position all of the sprites where the scroll will be when it is in its highest position. Then stretch the first and last line of each sprite so that the 19 sprite lines in the middle will be on the desired place. Opening the borders now is trivial, because all of the sprites are present on all of the scan lines and they steal a constant amount of time. However, we lose two sprite lines. We might not want to use the first and the last line for graphics, because they are stretched.
[See previous C=Hacking Issues for more information about stretching and
 stolen cycles.]

A more difficult approach is to unroll the routine and let another routine count the sprites present in each line and then change the time the routine uses accordingly. In this way you save time during the display for other effects, like color bars, because stretching will take at least 12 cycles on each raster line. On the other hand, if the sinus is constant (user is not allowed to change it), it is usually possible to embedd the count routine directly to the border opening part of the routine.

More sprites

You don't necassarily need to plot the characters in sprites to have more than eight characters. Using a sprite multiplexing techiques you can double or triple the number of sprites available. You can divide the scroll vertically into several areas and because the y-coordinate of the scroll is a sinus, there always is a fixed maximum number of sprites in each area. This number is always smaller than the total number of sprites in the whole scroll. I won't go into detail, but didn't want to leave this out completely. [See picture 3.]

Smoother and smoother

Why be satisfied with a scroll with only 40 different slices horizontally ? It should be possible to count own coordinates for each pixel column on the scroll. In fact the program won't be much different, but the routine must also mask the unwanted bits and write the byte to memory with ORA+STA. When you think about it more, it is obvious that this takes a generous amount of time, handling every bit seperately will take much more than eight times the time a simple LDA+STA takes. Some coders have avoided this by plotting the same character to different character sets simultaneously and then changing the charsets appropriately, but the resulting scroll won't be much larger than 96x32 pixels.

Picture 1 - Two character codes will make a graphical bitmap

Screen memory:

```
 ____________________________________
|Char   |Char   |Char   |Char   |  ...
|Code   |Code   |Code   |Code   |
|0      |2      |80     |80     | .
|       |**   **|       |       | .
|       |**   **|       |       | .
|*****  |****** |       |       |
|****** | ****  |       |       |
|**___**|__**___|_______|_______|
|**   **|  **   | ****  |Char   |
|**   **|  **   |****** |Code   |
|****** |       |**   **|6      |
|*****  |       |**     |       |
|Char   |Char   |**   **|       |
|Code   |Code   |****** |       |
|1      |3      |4****  |*****  |
|_______|_______|_______|******_|
|Char   |Char   |       |**   **|
|Code   |Code   |       |****** |
|80     |80     |       |*****  |
|       |       |       |**     |
|       |       |Char   |**ar   |
|       |       |Code   |Code   |
|       |       |5      |7      |
|_______|_______|_______|_______|
```

Character set memory:

```
 ________________________________________________________________
|Char 0 |Char 1 |Char 2 |Char 3 |Char 4 |Char 5 |Char 6 |Char 7 | ...
|_______|_______|_______|_______|_______|_______|_______|_______|__
      DDDDDDDD      YYYYYYYY     CCCCCCCC              PPPPPPPP
 First column    Second column   Third column    Fourth column
```

Picture 2 - DYCP with sprites

Sprite 0

```
 _______________________
|       |**   **|       |
|       |**   **|       |
|       |****** |       |
|*****  | ****  |       |
|****** |  **   |       |
|**   **|  **   |       |
|**   **|  **   |       |
|**___**|_______|_______|
|****** |       | ****  |
|*****  |       |****** |
|       |       |**   **|
|       |       |**     |
|       |       |**   **|
|       |       |****** |
|       |       | ****  |
|       |       |       | Sprite 1
```

```
|       |       |       | _______________________
|       |       |       ||*****  |       |       |
|       |       |       ||****** |       |       |
|       |       |       ||**  ** |       |       |
|_______|_______|_______||****** |       |       |
                         |*****  |       |       |
                         |**     |       |       |
                         |**     |       |       |
                         |_______|_______|_______|
                         |       |       |       |
                         |       |       |       |
                         |       |       |       |
                         |       | ****  |       |
                         |       | ****  |       |
                         |       |       |****** |
                         |       |       |****** |
                         |_______|_______|__**___|
                         |       |       |   **  |
                         |       |       |   **  |
                         |       |       |   **  |
                         |       |       |   **  |
                         |_______|_______|_______|
---------------------------------------------------------------------------
Picture 3 - Sprite multiplexing

                                __          Set coordinates for eight sprites
                             __|3 |         that start from the top half.
                            |4 |  |__
                          __|  `--|2 |
                         |5 `--'  |  |
                         |  |     `--'__
                         `--'        |1 |
                                     |  |
                       __            `--'
                      |6 |
                      |  |               __
                      `--'              |0 |
                                        |  |
-__-------------------------------------`--'When VIC has displayed the last
|0 |                __                      sprite, set coordinates for the
|  |               |6 |                     sprites in the lower half of the
`--'               |  |                     area.
                   `--'
    __
   |1 |         __
   |  |        |5 |
   `--'__      |  |
      |2 |   __`--'
      |  |__|4 |                            You usually have two sprites that
      `--|3 |  |                            are only 'used' once so that you
         |  `--'                            can change other sprites when VIC
         `__'                               is displaying them.
---------------------------------------------------------------------------
```

DYCP demo program (PAL)

```
SINUS=  $CF00   ; Place for the sinus table
CHRSET= $3800   ; Here begins the character set memory
GFX=    $3C00   ; Here we plot the dycp data
X16=    $CE00   ; values multiplicated by 16 (0,16,32..)
D16=    $CE30   ; divided by 16  (16 x 0,16 x 1 ...)
START=  $033C   ; Pointer to the start of the sinus
COUNTER= $033D  ; Scroll counter (x-scroll register)
POINTER= $033E  ; Pointer to the text char
YPOS=   $0340   ; Lower 4 bits of the character y positions
YPOSH=  $0368   ; y positions divided by 16
CHAR=   $0390   ; Scroll text characters, multiplicated by eight
ZP=     $FB     ; Zeropage area for indirect addressing
ZP2=    $FD
AMOUNT= 38      ; Amount of chars to plot-1
PADCHAR= 32     ; Code used for clearing the screen

*= $C000

        SEI             ; Disable interrupts
        LDA #$32        ; Character generator ROM to address space
        STA $01
        LDX #0
LOOP0   LDA $D000,X     ; Copy the character set
```

```
        STA CHRSET,X
        LDA $D100,X
        STA CHRSET+256,X
        DEX
        BNE LOOP0
        LDA #$37         ; Normal memory configuration
        STA $01
        LDY #31
LOOP1   LDA #66          ; Compose a full sinus from a 1/4th of a
        CLC              ;   cycle
        ADC SIN,X
        STA SINUS,X
        STA SINUS+32,Y
        LDA #64
        SEC
        SBC SIN,X
        STA SINUS+64,X
        STA SINUS+96,Y
        INX
        DEY
        BPL LOOP1
        LDX #$7F
LOOP2   LDA SINUS,X
        LSR
        CLC
        ADC #32
        STA SINUS+128,X
        DEX
        BPL LOOP2

        LDX #39
LOOP3   TXA
        ASL
        ASL
        ASL
        ASL
        STA X16,X        ; Multiplication table (for speed)
        TXA
        LSR
        LSR
        LSR
        LSR
        CLC
        ADC #>GFX
        STA D16,X        ; Dividing table
        LDA #0
        STA CHAR,X       ; Clear the scroll
        DEX
        BPL LOOP3
        STA POINTER      ; Initialize the scroll pointer
        LDX #7
        STX COUNTER
LOOP10  STA CHRSET,X     ; Clear the @-sign..
        DEX
        BPL LOOP10

        LDA #>CHRSET     ; The right page for addressing
        STA ZP2+1
        LDA #<IRQ        ; Our interrupt handler address
        STA $0314
        LDA #>IRQ
        STA $0315
        LDA #$7F         ; Disable timer interrupts
        STA $DC0D
        LDA #$81         ; Enable raster interrupts
        STA $D01A
        LDA #$A8         ; Raster compare to scan line $A8
        STA $D012
        LDA #$1B         ; 9th bit
        STA $D011
        LDA #30
        STA $D018        ; Use the new charset
        CLI              ; Enable interrupts and return
        RTS

IRQ     INC START        ; Increase counter
        LDY #AMOUNT
        LDX START
LOOP4   LDA SINUS,X      ; Count a pointer for each text char and according
        AND #7           ;  to it fetch a y-position from the sinus table
        STA YPOS,Y       ;   Then divide it to two bytes
```

```
        LDA SINUS,X
        LSR
        LSR
        LSR
        STA YPOSH,Y
        INX             ; Chars are two positions apart
        INX
        DEY
        BPL LOOP4

        LDA #0
        LDX #79
LOOP11  STA GFX,X       ; Clear the dycp data
        STA GFX+80,X
        STA GFX+160,X
        STA GFX+240,X
        STA GFX+320,X
        STA GFX+400,X
        STA GFX+480,X
        STA GFX+560,X
        DEX
        BPL LOOP11

MAKE    LDA COUNTER     ; Set x-scroll register
        STA $D016
        LDX #AMOUNT
        CLC             ; Clear carry
LOOP5   LDY YPOSH,X     ; Determine the position in video matrix
        TXA
        ADC LINESL,Y    ; Carry won't be set here
        STA ZP          ; low byte
        LDA #4
        ADC LINESH,Y
        STA ZP+1        ; high byte
        LDA #PADCHAR    ; First clear above and below the char
        LDY #0          ; 0. row
        STA (ZP),Y
        LDY #120        ; 3. row
        STA (ZP),Y
        TXA             ; Then put consecuent character codes to the places
        ASL             ;  Carry will be cleared
        ORA #$80        ; Inverted chars
        LDY #40         ; 1. row
        STA (ZP),Y
        ADC #1          ; Increase the character code, Carry won't be set
        LDY #80         ; 2. row
        STA (ZP),Y

        LDA CHAR,X      ; What character to plot ? (source)
        STA ZP2         ;  (char is already multiplicated by eight)
        LDA X16,X       ; Destination low byte
        ADC YPOS,X      ;  (16*char code + y-position's 3 lowest bits)
        STA ZP
        LDA D16,X       ; Destination high byte
        STA ZP+1

        LDY #6          ; Transfer 7 bytes from source to destination
        LDA (ZP2),Y : STA (ZP),Y
        DEY             ; This is the fastest way I could think of.
        LDA (ZP2),Y : STA (ZP),Y
        DEY
        LDA (ZP2),Y : STA (ZP),Y
        DEY
        LDA (ZP2),Y : STA (ZP),Y
        DEY
        LDA (ZP2),Y : STA (ZP),Y
        DEY
        LDA (ZP2),Y : STA (ZP),Y
        DEY
        LDA (ZP2),Y : STA (ZP),Y
        DEX
        BPL LOOP5       ; Get next char in scroll

        LDA #1
        STA $D019       ; Acknowledge raster interrupt

        DEC COUNTER     ; Decrease the counter = move the scroll by 1 pixel
        BPL OUT
LOOP12  LDA CHAR+1,Y    ; Move the text one position to the left
        STA CHAR,Y      ;  (Y-register is initially zero)
        INY
```

```
        CPY #AMOUNT
        BNE LOOP12
        LDA POINTER
        AND #63          ; Text is 64 bytes long
        TAX
        LDA SCROLL,X     ; Load a new char and multiply it by eight
        ASL
        ASL
        ASL
        STA CHAR+AMOUNT ; Save it to the right side
        DEC START        ; Compensation for the text scrolling
        DEC START
        INC POINTER      ; Increase the text pointer
        LDA #7
        STA COUNTER      ; Initialize X-scroll

OUT     JMP $EA7E        ; Return from interrupt

SIN     BYT 0,3,6,9,12,15,18,21,24,27,30,32,35,38,40,42,45
        BYT 47,49,51,53,54,56,57,59,60,61,62,62,63,63,63
                         ; 1/4 of the sinus

LINESL  BYT 0,40,80,120,160,200,240,24,64,104,144,184,224
        BYT 8,48,88,128,168,208,248,32

LINESH  BYT 0,0,0,0,0,0,0,1,1,1,1,1,1,2,2,2,2,2,2,2,3

SCROLL  SCR "THIS@IS@AN@EXAMPLE@SCROLL@FOR@"
        SCR "COMMODORE@MAGAZINE@BY@PASI@OJALA@@"
                         ; SCR will convert text to screen codes
```

Basic loader for the Dycp demo program (PAL)

```
1 S=49152
2 DEFFNH(C)=C-48+7*(C>64)
3 CH=0:READA$,A:PRINTA$:IFA$="END"THENPRINT"<white><clr>":SYS49152:END
4 FORF=0TO31:Q=FNH(ASC(MID$(A$,F*2+1)))*16+FNH(ASC(MID$(A$,F*2+2)))
5 CH=CH+Q:POKES,Q:S=S+1:NEXT:IFCH=ATHEN3
6 PRINT"CHECKSUM ERROR":END
100 DATA 78A9328501A200BD00D09D0038BD00D19D0039CAD0F1A9378501A01FA942187D, 3441
101 DATA 75C19D00CF9920CFA94038FD75C19D40CF9960CFE88810E4A27FBD00CF4A1869, 4302
102 DATA 209D80CFCA10F3A2278A0A0A0A0A9D00CE8A4A4A4A4A18693C9D30CEA9009D90, 3231
103 DATA 03CA10E58D3E03A2078E3D039D0038CA10FAA93885FEA99B8D1403A9C08D1503, 3338
104 DATA A97F8D0DDCA9818D1AD0A9A88D12D0A91B8D11D0A91E8D18D05860EE3C03A026, 3864
105 DATA AE3C03BD00CF2907994003BD00CF4A4A4A996803E8E88810EAA900A24F9D003C, 3256
106 DATA 9D503C9DA03C9DF03C9D403D9D903D9DE03D9D303ECA10E5AD3D038D16D0A226, 3739
107 DATA 18BC68038A7995C185FBA90479AAC185FCA920A00091FBA07891FB8A0A0980A0, 4224
108 DATA 2891FB6901A05091FBBD900385FDBD00CE7D400385FBBD30CE85FCA006B1FD91, 4440
109 DATA FB88B1FD91FB88B1FD91FB88B1FD91FB88B1FD91FB88B1FD91FB88B1FD91FBCA, 6225
110 DATA 109FEE19D0CE3D031028B99103999003C8C026D0F5AD3E03293FAABDBFC10A0A, 3593
111 DATA 0A8DB603CE3C03CE3C03EE3E03A9078D3D034C7EEA000306090C0F1215181B1E, 2159
112 DATA 202326282A2D2F313335363839 3B3C3D3E3E3F3F3F00285078A0C8F018406890, 2268
113 DATA B8E008305880A8D0F8200000000000000000010101010101020202020202020314, 1379
114 DATA 08091300091300010E000518010D100C05001303120F0C0C00060F1200030F0D, 304
115 DATA 0D0F040F1205000D0107011A090E0500021900100113090000F0A010C01000000, 257
200 DATA END,0
```

Uuencoded C64 executable of the basic loader (PAL)

```
begin 644 dycp.64
M`0@-"`$`4[(T.3$U,@`F"`(`EJ5(*$,ILD.K-#BJ-ZPH0[$V-"D`4@@#`$-(@
MLC`ZAT$D+$$ZF4$D.HM!)+(B14Y$(J>9(@63(CJ>-#DQ-3(Z@`")"`0`@4:RE
M,*0S,3I1LJ5(*,8HRBA!)"Q&K#*J,2DI*:PQ-JJE2"C&*,HH020L1JPRJC(IA
M*2D`J@@%`$-(LD-(JE$ZEU,L43I3LE.J,3J".HM#2+)!IS,`P@@&`)DB0TA%.
M0TM354T@15)23U(B.H``#PED`(,@-SA!.3,R.#4P,4$R,#!"1#`P1#`Y1#`P.
M,SA"1#`P1#$Y1#`P,SE#040P1C%!.3,W.#4P,4$P,49!.30R,3@W1"P@,S0T#
M,0!<"64`@R`W-4,Q.40P,$-&.3DR,$-&03DT,#,X1D0W-4,Q.40T,$-&.3DV&
M,$-&13@X.#$P131!,C=&0D0P,$-&-$$Q.#8Y+"`T,S`R`*D)9@"#(#(P.40X3
M,$-&0T$Q,$8S03(R-SA!,$$P03!!,$$Y1#`P0T4X031!-$$T031!,3@V.3-#P
M.40S,$-%03DP,#E$.3`L(#,R,S$`]@EG`(,@,#-#03$P134X1#-%,#-!,C`WY
M.$4S1#`S.40P,#,X0T$Q,$9!03DS.#@U1D5!.3E".$0Q-#`S03E#,#A$,34P@
M,RP@,S,S.`!#"F@`@R!!.3=&.$0P1$1#03DX,3A$,4%$,$$Y03@X1#$R1#`P!!B
M.3%".$0Q,40P03DQ13A$,3A$,#4X-C!%13-#,#-!,#(V+"`S.#8T`)`*:0"#]
M($%%,T,P,T)$,#!#1C(Y,#<Y.30P,#-"1#`P0T8T031!-$$Y.38X,#-%.$4X$
M.#@Q,$5!03DP,$$R-$8Y1#`P,T,L(#,R-38`W0IJ`(,@.40U,#-#.40U,#-]
M.41&,#-#.40T,#-$.40Y,#-$.41%,#-$.40S,#-%0T$Q,$4U040S1#`S.$0Q*
M-D0P03(R-BP@,S<S.0`J"VL`@R`Q.$)#-C@P,SA!-SDY-4,Q.#5&0D$Y,#0W^
M.4%!0S$X-49#03DR,$$P,#`Y,49"03`W.#DQ1D(X03!!,#DX,$$P+"`T,C(T:
M``<+;`"#(#(X.3%&0C8Y,#%!,#4P.3%&0D)$.3`P,S@U1D1"1#`P0T4W1#0P;
```

```
M,#,X-49"0D0S,$-%.#5&0T$P,#9",49$.3$L(#0T-#``Q`MM`(,@1D(X.$(Q?
M1D0Y,49".#A",49$.3%&0C@X0C%&1#DQ1D(X.$(Q1D0Y,49".#A",49$.3%&V
M0C@X0C%&1#DQ1D)#02P@-C(R-0`1#&X`@R`Q,#E&144Q.40P0T4S1#`S,3`RK
M.$(Y.3$P,SDY.3`P,T,X0S`R-D0P1C5!1#-%,#,R.3-&04%"1$)&0S$P03!!M
M+"`S-3DS`%X,;P"#(#!!.$1"-C`S0T4S0S`S0T4S0S`S144S13`S03DP-SA$H
M,T0P,S1#-T5%03`P,#,P-C`Y,$,P1C$R,34Q.#%",44L(#(Q-3D`JPQP`(,@0
M,C`R,S(V,C@R03)$,D8S,3,S,S4S-C,X,SDS0C-#,T0S13-%,T8S1C-&,#`R[
M.#4P-SA!,$,X1C`Q.#0P-C@Y,"P@,C(V.`#X#'$`@R!".$4P,#@S,#4X.#!!8
M.$0P1C@R,#`P,#`P,#`P,#`P,#$P,3`Q,#$P,3`Q,#(P,C`R,#(P,C`R^
M,#(P,S$T+"`Q,S<Y`$0-<@"#(#`X,#DQ,S`P,#DQ,S`P,#$P13`P,#4Q.#`Q7
M,$0Q,#!#,#4P,#$S,#,Q,C!&,$,P0S`P,#8P1C$R,#`P,S!&,$0L(#,P-`"02
M#7,`@R`P1#!&,#0P1C$R,#4P,#!$,#$P-S`Q,4$P.3!%,#4P,#`R,3DP,#$P.
L,#$Q,S`Y,#`P1C!!,#$P0S`Q,#`P,#`P+"`R-3<`G`W(`(,@14Y$+#`````PK
``

end
size 1439
--------------------------------------------------------------------------
Uuencoded C64 executable of the basic loader (NTSC)

begin 644 dycp-ntsc.bas
M`0@-"`$`4[(T.3$U,@`F"`(`EJ5(*$,ILD.K-#BJ-ZPH0[$V-"D`40@#`$-(?
MLC`ZAT$D+$$ZF4$D.HM!)+(B14Y$(J>9(I,B.IXT.3$U,CJ``(@(!`"!1K(P/
MI#,Q.E&RI4@HQBC**$$D+$:L,JHQ*2DIK#$VJJ5(*,8HRBA!)"Q&K#*J,BDI:
M*0"I"`4`0TBR0TBJ43J74RQ1.E.R4ZHQ.H(ZBT-(LD&G,P#!"`8`F2)#2$5#F
M2U-532!%4E)/4B(Z@`#'"%H`@``4"60`@R`W.$$Y,S(X-3`Q03(P,$)$,#!$L
M,#E$,#`S.$)$,#!$,3E$,#`S.4-!1#!&,$Y,S<X-3`Q03`Q1D$Y-#(Q.#=$I
M+"`S-#0Q`&$)90"#(#<U0S$Y1#`P0T8Y.3(P0T9!.30P,SA&1#<U0S$Y1#0P!
M0T8Y.38P0T9%.#@X,3!%-$$R-T9"1#`P0T8T03$X-CDL(#0S,#(`K@EF`(,@R
M,C`Y1#@P0T9#03$P1C-!,C(W.$$P03!!,$$P03E&,#!!-$$T031!-$$Q.J
M.#8Y,T,Y1#,P0T5!.3`P.40Y,"P@,S(S,0#["6<`@R`P,T-!,3!%-3A$,T4P.
M,T$R,#<X13$$,#,Y1#`P,SA#03$P1D%!.3`P.#5&14$Y.4(X1#$T,#-!.4,P;
M.$0Q-3`S+"`S,S,X`$@*:`"#($$Y-T8X1#!$,1$-!.3@Q.$0Q-40P03E",#A$:
M,3)$,$$Y,4(X1#$Q1#!!.3%%.$0Q,$0P-3@V,$5%,T,P,T4P,$1#,4(X,$C(`9
ME0II`(,@044S0S`S0D0P,$-&,$-DP-SDY-#`P,T)$,#!#1C1!-$$T03DY-C@PB
M,T4X13@X.#$P14%!.3`P03(T1CE$,#`S0RP@,S(U-@#B"FH`@R`Y1#4P,T,Y$
M1$$P,T,Y1$8P,T,Y1#0P,T0Y1#DP,T0Y1$4P,T0Y1#,P,T5#03$P135!1#-$E
M,#,X1#$V1#!!,C%#+"`S-S(Y`"\+:P"#(#$X0D,V.#`S.$$W.3DU0S$X-49")
M03DP-#<Y04%#,3@U1D-!.3(P03`P,#DQ1D)!,#<X.3%&0CA!,$$P.3@P03`L#
M(#0R,C0`?`ML`(,@,C@Y,49"-CDP,4$P-3`Y,49"0D0Y,#`S.#5&1$)$,$,!#H
M13=$-#`P,S@U1D)"1#,P,T4X-49#03`P-D(Q1D0Y,2P@-#0T,`#)"VT`@R!&C
M0C@X0C%&1#DQ1D(X.$(Q1D0Y,49".#A",49$.3%&0C@X0C%&1#DQ1D(X.$(QA
M1D0Y,49".#A",49$.3%&0D-!+"`V,C(U`!8,;@"#(#$P.49%13$Y1#!!#13-$T
M,#,Q,#(X0CDY,3`S.3DY,#`S0SA#,#(V1#!&-4%$,T4P,S(Y,T9!04)$0D9#0
M,3!!,$$L(#,U.3,`8PQO`(,@,$$X1$$V-#,P,#-#,#-#,#-%,3-%,#-!D
M.3`W.$0S1#`S-$$W145!,#`P,S`V,#DP0S!&,3$S-$$X,#4Q,$$T.4,P"P2
M#'``@R`R,#(S,C8R.#)!,D0R1C,Q,S,S-3,V,S@S.3-",T,S1#-%,T4S1C-&/
M,T8P,#(X-3`S0SA&,$$X-#DP+"`R,C`Y`+`,<0"#($$X13`P13`P,#8P,$$P.P2
M-3@X,$$X1#!&.(P,#`P,#`P,#`P,#`P,3`Q,#$P,3`Q,#(P,C`R,#(P>
M,C`R,#(P,C`S,30L(#$S-SD`20UR`(,@,#@P.3$S,#`P,3$S,#`P,3!%,#`P3
M-3$X,#$P,$$P,#$P,S$P,C$P,$8P,S!#,#$P-C!&,3!0S$P,#`P,3`Q,#$P@J
M,S`T`)4-<P"#(#!!$$,$8P-#!!&,3(P-3`P,3`W,#$Q03`Y,$4P-3`P,3(Q`
M.3`P,3$S,3`P0S$P,#$P,#`P,#`P+#`L(#(U-P"A#7@`@R!%3D0LZ
$,````#`P0
``

end
size 1444
```

===

Opening the borders

by Pasi 'Albert' Ojala (po87553@cs.tut.fi or albert@cc.tut.fi)
Written: 20-Jul-92

All timings are in PAL, principles will apply to NTSC too.
Refer to VIC memory map in Hacking Issue 4.

VIC has many features and transparent borders are one of them. You can not make characters appear in the border, but sprites are displayed in the border too. "How to do this then?" is the big question.

The screen resolution in C64 has been and will be 320 x 200 pixels. Most games need to use the whole screen to be efficient or just plain playable. But there still is that useless border area, and you can put score and other status information there instead of having them interfere with the full-screen smooth-scrolling playing area.

How to disable the vertical borders

When VIC (Video Interface Controller) has displayed all character rows, it will start displaying the vertical border area. It will start displaying the characters again in top of the screen. The row select register sets the number of character lines on the screen. If we select the 24-row display when VIC is drawing the last (25th) row, it does not start to draw the border at all ! VIC will think that it already started to draw the border.

The 25-row display must be selected again in the top of the screen, so that the border may be opened in the next frame too. The number of displayed rows can be selected with the bit 3 in $d011. If the bit is set, VIC will display 25 rows and 24 rows otherwise. We have to clear the bit somewhere during the last row (raster lines $f2-$fa) and set it again in top of the screen or at least somewhere before the last row (line $f2). This has to be done in every frame (50 times per second in PAL).

How to open the sideborders

The same trick can be applied to sideborders. When VIC is about to start displaying the sideborder, just select 38-column mode and restore 40-column mode so that you can do the trick again in the next scan line. If you need to open the sideborders in the bottom or top border area, you have to open the vertical borders also, but there shouldn't be any difficulty in doing that.

There is two drawbacks in this. The timing must be precise, one clock cycle off and the sideborder will not open (the sprites will generally take care of the timing) and you have to do the opening on each and every line. With top/bottom borders once in a frame was enough.

Another problem is bad-lines. There is not enough time to open the borders during a bad line and still have all of the sprites enabled. One solution is to open the borders only on seven lines and leave the bad lines unopened. Another way is to use less than eight sprites. You can have six of them on a bad line and still be able to open the sideborders (PAL). The old and still good solution is to scroll the bad lines, so that VIC will not start to draw the screen at all until it is allowed to do so.
[Read more about bad lines from previous C=Hacking Issues]

Scrolling the screen

VIC begins to draw the screen from the first bad line. VIC will know what line is a bad line by comparing its scan line counter to the vertical scroll register : when they match, the next line is a bad line. If we change the vertical scroll register ($d011), the first bad line will move also. If we do this on every line, the line counter in VIC will never match with it and the drawing never starts (until it is allowed to do so).

When we don't have to worry about bad lines, we have enough time to open the borders and do some other effects too. It is not necassary to change the vertical scroll on every line to get rid of the bad lines, just make sure that it never matches the line counter (or actually the least significant 8 bits).

You can even scroll the bad lines independently and you have FLD - Flexible Line Distance. You just allow a bad line when it is time to display the next character row. With this you can bounce the lines or scroll a hires picture very fast down the screen. But this has not so much to do with borders, so I will leave it to another article. (Just send requests and I might start writing about FLD ..)

Garbage appearing

When we open the top and bottom borders, some graphics may appear. Even though VIC has already completed the graphics data fetches for the screen area, it will still fetch data for every character position in top and bottom borders. This will not do any harm though, because it does not generate any bad lines and happens during video fetch cycles [see Missing Cycles article]. VIC reads the data from the last address in the current video bank, which is normally $3fff and displays this over and over again.

If we change the data in this address in the border area, the change will be visible right away. And if you synchronize the routine to the beam position, you can have a different value on each line. If there is nothing else to do in the border, you can get seven different values on each scan line.

The bad thing about this graphics is that it is impossible to change its color - it is always black. It is of course possible to use inverted graphics and change the background color. And if you have different data on each line, you can as easily have different color(s) on each line too.

If you don't use $3fff for any effects, it is a good idea to set it to zero, but remember to check that you do not store anything important in that address. In one demo I just cleared $3fff and it was right in the middle of another packed demopart. It took some time to find out what was wrong with the other part.

Horizontal scrolling

This new graphics data also obeys the horizontal scroll register ($D016), so you can do limited tech-tech effects in the border too. You can also use sprites and open the sideborders. You can see an example of the tech-tech effect in the first example program. Multicolor mode select has no effect on this data. You can read more about tech-tech effects in a future article.

Example routine

The example program will show how to open the top and bottom borders and how to use the $3fff-graphics. It is fairly well commented, so just check it for details. The program uses a sprite to do the synchronization [see Missing Cycles article] and reads a part of the character ROM to the display data buffer. To be honest, I might add that this is almost the same routine than the one in the Missing Cycles article. I have included both PAL and NTSC versions of the executables.

The example program - $3fff-graphics

```
IMAGE0= $CE00   ; First graphics piece to show
IMAGE1= $CF00   ; Second piece
TECH=   $CD00   ; x-shift
RASTER= $FA     ; Rasterline for the interrupt
DUMMY=  $CFFF   ; Dummy-address for timing (refer to missing_cycles-article)

*= $C000
        SEI             ; Disable interrupts
        LDA #$7F        ; Disable timer interrupts (CIA)
        STA $DC0D
        LDA #$01        ; Enable raster interrupts (VIC)
        STA $D01A
        STA $D015       ; Enable the timing sprite
        LDA #<IRQ
        STA $0314       ; Interrupt vector to our routine
        LDA #>IRQ
        STA $0315
        LDA #RASTER     ; Set the raster compare (9th bit will be set
        STA $D012       ;  inside the raster routine)
        LDA #RASTER-20  ; Sprite is situated 20 lines before the interrupt
        STA $D001

        LDX #111
        LDY #0
        STY $D017       ; Disable y-expand
        LDA #$32
        STA $01         ; Select Character ROM
LOOP0   LDA $D000,X
        STA IMAGE0,Y    ; Copy a part of the charset to be the graphics
        STA IMAGE0+112,Y
        LDA $D800,X
        STA IMAGE1,Y
        STA IMAGE1+112,Y
        INY             ; Until we copied enough
        DEX
        BPL LOOP0
        LDA #$37        ; Char ROM out of the address space
        STA $01

        LDY #15
LOOP1   LDA XPOS,Y      ; Take a half of a sinus and mirror it to make
        STA TECH,Y      ;  a whole cycle and then copy it as many times
        STA TECH+32,Y   ;   as necassary
        LDA #24
        SEC
        SBC XPOS,Y
        STA TECH+16,Y
        STA TECH+48,Y
        DEY
        BPL LOOP1
        LDY #64
LOOP2   LDA TECH,Y
        STA TECH+64,Y
        STA TECH+128,Y
```

```
        DEY
        BPL LOOP2
        CLI             ; Enable interrupts
        RTS             ; Return to basic (?)


IRQ     LDA #$13        ; Open the bottom border (top border will open too)
        STA $D011
        NOP
        LDY #111        ; Reduce for NTSC ?
        INC DUMMY       ; Do the timing with a sprite
        BIT $EA         ; Wait a bit (add a NOP for NTSC)

LOOP3   LDA TECH,Y      ; Do the x-shift
        STA $D016
FIRST   LDX IMAGE0,Y    ; Load the graphics to registers
SECOND  LDA IMAGE1,Y
        STA $3FFF       ; Alternate the graphics
        STX $3FFF
        STA $3FFF
        STX $3FFF
        STA $3FFF
        STX $3FFF
        STA $3FFF
        STX $3FFF
        STA $3FFF
        STX $3FFF
        LDA #0          ; Throw away 2 cycles (add a NOP for NTSC)
        DEY
        BPL LOOP3

        STA $3FFF       ; Clear the graphics
        LDA #8
        STA $D016       ; x-scroll to normal
        LDA #$1B
        STA $D011       ; Normal screen (be ready to open the border again)
        LDA #111
        DEC FIRST+1     ; Move the graphics by changing the low byte of the
        BPL OVER        ;  load instruction
        STA FIRST+1
OVER    SEC
        SBC FIRST+1
        STA SECOND+1    ; Another graphics goes to opposite direction
        LDA LOOP3+1     ; Move the x-shift also
        SEC
        SBC #2
        AND #31         ; Sinus cycle is 32 bytes
        STA LOOP3+1

        LDA #1
        STA $D019       ; Acknowledge the raster interrupt
        JMP $EA31       ; jump to the normal irq-handler

XPOS    BYT $C,$C,$D,$E,$E,$F,$F,$F,$F,$F,$F,$F,$E,$E,$D,$C
        BYT $C,$B,$A,$9,$9,$8,$8,$8,$8,$8,$8,$8,$9,$9,$A,$B
                        ; half of the sinus

---------------------------------------------------------------------------
Basic loader for the $3fff-program (PAL)

1 S=49152
2 DEFFNH(C)=C-48+7*(C>64)
3 CH=0:READA$,A:PRINTA$:IFA$="END"THENPRINT"<clear>":SYS49152:END
4 FORF=0TO31:Q=FNH(ASC(MID$(A$,F*2+1)))*16+FNH(ASC(MID$(A$,F*2+2)))
5 CH=CH+Q:POKES,Q:S=S+1:NEXT:IFCH=ATHEN3
6 PRINT"CHECKSUM ERROR":END
100 DATA 78A97F8D0DDCA9018D1AD08D15D0A9718D1403A9C08D1503A9FA8D12D0A9E68D,4003
101 DATA 01D0A26FA0008C17D0A9328501BD00D09900CE9970CEBD00D89900CF9970CFC8,4030
102 DATA CA10EAA9378501A00FB9DCC09900CD9920CDA91838F9DCC09910CD9930CD8810,4172
103 DATA E8A040B900CD9940CD9980CD8810F45860A9138D11D0EAA06FEEFFCF24EAB906,4554
104 DATA CD8D16D0BE53CEB91CCF8DFF3F8EFF3F8DFF3F8EFF3F8DFF3F8EFF3F8DFF3F8E,4833
105 DATA FF3F8DFF3F8EFF3FA9008810D18DFF3FA9088D16D0A91B8D11D0A96FCE85C010,4163
106 DATA 038D85C038ED85C08D88C0AD7FC018E901291F8D7FC0EE19D04C31EA0C0C0D0E,3719
107 DATA 0E0F0F0F0F0F0F0F0E0E0D0C0C0B0A0909080808080808080809090A0B00000000,318
200 DATA END,0

---------------------------------------------------------------------------
An uuencoded C64 executable $3fff-program (PAL)

begin 644 xFFF.64
M`0@-"`$`4[(T.3$U,@`F"`(`EJ5(*$,ILD.K-#BJ-ZPH0[$V-"D`40@#`$-(?
```

```
MLC`ZAT$D+$$ZF4$D.HM!)+(B14Y$(J>9(I,B.IXT.3$U,CJ``(@(!`"!1K(P/
MI#,Q.E&RI4@HQBC**$$D+$:L,JHQ*2DIK#$VJJ5(*,8HRBA!)"Q&K#*J,BDI:
M*0"I"`4`0TBR0TBJ43J74RQ1.E.R4ZHQ.H(ZBT-(LD&G,P#!"`8`F2)#2$5#F
M2U-532!%4E)/4B(Z@``."60`@R`W.$$Y-T8X1#!$1$-!.3`Q.$0Q040P.$0QK
M-40P03DW,3A$,30P,T$Y0S`X1#$U,#-!.49!.$0Q,D0P03E%-CA$+"`T,#`S9
M`%L)90"#(#`Q1#!!,C9&03`P,#A#,3=$,$$Y,S(X-3`Q0D0P,$0P.3DP,$-%Y
M.3DW,$-%0D0P,$0X.3DP,$-&.3DW,$-&0S@L(#0P,S``J`EF`(,@0T$Q,$5!S
M03DS-S@U,#%!,#!&0CE$0T,P.3DP,$-$.3DR,$-$03DQ.#,X1CE$0T,P.3DQL
M,$-$.3DS,$-$.#@Q,"P@-#$W,@#U"6<`@R!%.$$P-#!".3`P0T0Y.30P0T0Y0
M.3@P0T0X.#$P1C0U.#8P03DQ,SA$,3%$,$5!03`V1D5%1D9#1C(T14%".3`V5
M+"`T-34T`$(*:`"#($-$.$0Q-D0P0D4U,T-%0CDQ0T-&.$1&1C-&.$5&1C-&%
M.$1&1C-&.$5&1C-&.$1&1C-&.$5&1C-&.$1&1C-&.$4L(#0X,S,`CPII`(,@'
M1D8S1CA$1D8S1CA%1D8S1D$Y,#`X.#$P1#$X1$9&,T9!.3`X.$0Q-D0P03DQ-
M0CA$,3%$,$$Y-D9#13@U0S`Q,"P@-#$V,P#<"FH`@R`P,SA$.#5#,#,X140X+
M-4,P.$0X.$,P040W1D,P,3A%.3`Q,CDQ1CA$-T9#,$5%,3E$,#1#,S%%03!#.
M,$,P1#!%+"`S-S$Y`"@+:P"#(#!%,$8P1C!&,$8P1C!&,$8P13!%,$0P0S!#P
M,$(P03`Y,#DP.#`X,#@P.#`Y,#DP03!",#`P,#`L(#,Q.``T>
-"\@`@R!%3D0L````#@P1
``

end
size 823
```

An uuencoded C64 executable $3fff-program (NTSC)

```
begin 644 xfff-ntsc.64
M`0@-"`$`4[(T.3$U,@`F"`(`EJ5(*$,ILD.K-#BJ-ZPH0[$V-"D`40@#`$-(?
MLC`ZAT$D+$$ZF4$D.HM!)+(B14Y$(J>9(I,B.IXT.3$U,CJ``(@(!`"!1K(P/
MI#,Q.E&RI4@HQBC**$$D+$:L,JHQ*2DIK#$VJJ5(*,8HRBA!)"Q&K#*J,BDI:
M*0"I"`4`0TBR0TBJ43J74RQ1.E.R4ZHQ.H(ZBT-(LD&G,P#!"`8`F2)#2$5#F
M2U-532!%4E)/4B(Z@`#'"%H`@``4"60`@R`W.$$Y-T8X1#!$1$-!.3`Q.$0QX
M040P.$0Q-40P03DW,3A$,30P,T$Y0S`X1#$U,#-!.49!.$0Q,D0P03E%-CA$H
M+"`T,#`S`&$)90"#(#`Q1#!!,C9&03`P,#A#,3=$,$$Y,S(X-3`Q0D0P,$0PX
M.3DP,$-%.3DW,$-%0D0P,$0X.3DP,$-&.3DW,$-&0S@L(#0P,S``K@EF`(,@H
M0T$Q,$5!03DS-S@U,#%!,#!&0C$U,P.3DP,$-$.3DR,$-$03DQ.#,X1CE$`
M14,P.3DQ,$-$.3DS,$-$.#@Q,"P@-#$W-@#["6<`@R!%.$$P-#!".3`P0T0Y8
M.30P0T0Y.3@P0T0X.#$P1C0U.#8P03DQ,SA$,3%$,$5!03`V1D5%1D9#1C(T+
M14%%04(Y+"`T-S@R`$@*:`"#(#`P0T0X1#$V1#!!,#!"13`P0T8X1$9&,T9&>
M,T8X14149&,T9&,T8X1$9&,T9&,T8X14149&,T9&,T8X1$9&,T8L(#0U.#``?
ME0II`(,@.$5&1C-&.$1&1C-&.$5&1C-&14%!.3`P.#@Q,$0P.$1&1C-&03DP`
M.#A$,39$,$$Y,4(X1#$Q1#!!.39&0T4X-4BP@-#,S,0#B"FH`@R!#,#,X1#XY
M1#@V0S`S.$5$.#5#,#@T.#A#,$%$.T1#,#$X13DP,3(Y,48X1#@P0T0!%13$YO
M1#`T0S,Q14$P.3`U`"X+:P"#(#!$,$4P1C!&,$8P1C!&,$8P13!%,$8P1C!&W
M,$4P13!$,$,P0S`Y,#@P.#`X,#@P.3`Y,$$P.3`Y,$$P.#`Y,$$P0D$R,#`L%
4(#4P-P`[";P`@R!%3D0L+3$````PB
``

end
size 830
```

==

A Heavy-Duty Power Supply for the C-64

by John C. Andrews (no email address)

As a Commodore User for the last 4 plus years, I am aware of the many articles and letters in the press that have bemoaned the burn-out problem of the C-64 power supply. When our Club BBS added a one meg drive and stayed on around the clock, the need for heavy-duty power supply became very apparent.... Three power supplies went in 3 successive days!

Part of the problem was my ignoring the seasons. You see during the winter I had set the power supply between the window and the screen, Yes, outside! With the advent of Spring... well, you get the picture.

The turn-around time forgetting a new commerical supply was not in the best interest of the BBS and its members. Therefore, taking power supply inhand, I proceeded to cut one open on my shop bandsaw. I do not suggest that you do this. The parts are FIRMLY and COMPLETELY encased in a hard plastic potting compound. The purpose of this is not to make the item difficult to repair, but to make the entire unit conductive to the heat generated inside. I doubt the wisedom of potting the fuse as well. However, CBM was probably thinking of the number of little fingers that could fit into an accessable fuse holder. if you want the punch line it is: the final circuit board and its componets are about the size of a box of matches. This includes the built-in metal heat sink.

From these minscule innards I traced out the circuit and increased the size of ALL components.

The handiest source of electronic parts is, of course, Radio Shack. All but one part can be purchased there.

212-1013	Capacitor, 35V, 4700 mF
212-1022	Capacitor, 35V, 10 uF
273-1515	Transformer, 2 Amp, 9-0-9 VAC
276-1184	Rectifier
270- 742	Fuse Block
270-1275	Fuses

Note that there are only five parts. The rest are fuses, fuse blocks, heat sinks, wire and misc. hardware. Note also that I have not listed any plugs and cords. This because you can clip the cords off of both sides of your defunct power supply. This will save you the hassle of wriing the DIN power plug correctly:

DIN PIN OUT		COLOR
pin 6	9VAC	black
pin 7	9VAC	black
pin 5	+5 Volts	blue
pin 1,2,3	shield, gnd	orange

The part that you can NOT get at Radio Shack is the power regulator. This part will have to be scrounged up from some local, big electronics supply house:

SK 9067	5 volt voltage regulator, 3+ amps. (I prefer the 5 amp.)

Radio Shack does carry regulators, but their capacity is no larger than that with which you started.

The Heat sinks, (yes, more than one!) are the key to the success of this project. The ones I used came from my Model Railroading days. Sorry to say, I did just ahve them 'lying about'. The heat sinks that I priced at the local electronics supply were more costly than the other parts. The worst case situation is that you may need to drill out a couple pieces of aluminum sheet. Try for 12 x 12, and bend them into square bottomed U-shapes to save room. heat sinks should not touch, or be electronically grounded to each other. You can also mount them on stand-offs from your chassis for total air circulation.

The Radio Shack transformer is rated at only 2 amps. If you can not find one with a higher rating elsewhere, it is possible to hook two in parallel to get a 4 ampere output. This si tricky, as it can be done either right or wrong!

Here is how to do it the right way:
  Tape off one yellow secondary lead on each transformer. With tape mark the four remaining secondary leads and letter them A and B on one transformer, C and D onthe other. Hook up the black primary leads to a plug to your 120 wall outlet:

```
                                  |-------------
 Note: *'s - indicate connections |            3 ||
       +'s - indicate skip overs  |            3 || (Transformer)
                                  |            3 ||
                                  |            3 ||
                                  |    ----------
                                  |    |
        +--\  /-------------------*----+---------
      --|120|/                         |       3 ||
      --|Vlt|            ____          |       3 ||
       -|Plg|------------|FUSE|--------*       3 ||
        +--/             ----          |       3 || (Transformer)
                                       |---------
```

  This would now be a good time to install a fuse in your 120 VAC line. Now before plugging this into the wall, tie two of the scondary leads (one from EACH transformer) together.

  Something like this: A--Xfmr--B+C--Xfmr--D

  Plug in your 120V side. Now using a VOM meter, measure the voltage between A and D.
    If the meter reads 18 volts, then:
      1. unplug from the 120.
      2. tie A and C together. tie B and D together.
      3. your 2 transformers will now give you 9 volts at 4 amps.
    If the meter reads 0 volts, then:
      1. unplug from the 120.
      2. tie A and D together. Tie B and C together.
      3. your 2 transformers will now give you 9 volts at 4 amps.

Below is the file corresponding to the full schematic of the power supply. [Ed's note: in GeoPaint format, converted by convert 2.5, then uuencoded]. As you can see in the picture, I used only one transformer. Because it got hot, I epoxied a small heat sink to it. While this solved the heat problem, it did not increase the capacity of the total power supply.

Note that I used fuses on all lines.

```
begin 700 schematic.
M@PD,<V-H96UA=&ECH*"@H*"@H`D&`0=8!P8-,Q(`4%)'(&9O<FUA='1E9"!'
M14]3(&9I;&4@5C$N,``CF````"```.``0X(``$5P<V]N($U8+3@P`*"@H*``
MUH>-UH?(F!AE`H4"D`+F`V"@H*"@H*"@H`@%`0A6!`<,`"````"""@A30TA%
M34%424.@H*"@H*"@````````````$P!"3$%35$52)U,@0T].5D525$52(%8R
M+C4$6`<&#2X"````@RP#0T].5D525*"@H*"@H*"@H"P2``99`Q$&`10`````
M```````````````````````````````````````#%;_____```.@``6?__F5
M55F:JJF555F:JJF555F:JJF555F:JJF?__F@``7```/___\```````-__[:`
M`/Y__[R#!P$``/__``!086EN="!);6%G92!6,2XQ````````````````````
M````````````9V5O4&%I;G0@("`@5C(N,``````@*$')!M`"J1*-(D"I`(TG
M0"#]/Y`%J0"-)T`@#!\@2$&I`(VL7ZD`A1&I!X40J3^%%ZGQA1:I7X4-J:R%
M#*!`J0X@3S&B_Z4"R0+P)LG_T`8@-$&X4*G)!M`-H$.I)B!/,2"APKA0F*VL
M7_`&(+T\(($\8%!A:0#_"@#_`3P!>`)+`EL"/P&R`38!B@%Z`38!C@(G`/\`
M_P#_`/\`_P#_`/\`_P#_`/\`_P#_`/\`_P#_`/\`_P#_`/\`_P#_`/\`_P#_
M`/\`_P#_`/\`_P#_`/\`````````````````````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M`````````````````````````````````````````````````````````/\`
M_P#_`/\`_P#_`/\`E0`"/R!%````````_P"%``,0_Q!$````````_P"&``+X
M"/\`_P"B`/^_H;\``"`@,!@,_PP810``````_P``"!`0,&#@_\#`0P``````
M_P``A1`#\!`0H`"("/\`_P#*``(@/X8``>#_`+```3"'(*@``F`PAA"8`(@0
MH`"("/\`_P"B`/^_H;\`(*@`B!"8`(00!!\0$!!$`````/\```"$"`3X"`@(
M_P#_`,(``O__A@`"X."&((L`#0$!`0($@("`#A$0$!.%``-98D.%``.,4MZ%
M``/@D)"0`$'0P`````````#_AP`!@,\`B""H_P#.``H#1$0HA0`##03$A0`#
M@("9AP`!!*``B""H`(@0H`"("/\`_P#_`.,``'2D1$1$0```).0$),0```"E
MI*2DF`````2HJ%!0HP"&(`)@8*(``1^$``P!$!#_````/-,``/B$``&`F`"(
M"/\`_P"B`/^_H;\``+```3"'(*@``F`PAA"8`(@0H`"("/\`_P"B`/^_H;\`
M(*@`B!"8`(00!!\0$!!$`````/\```"$"`3X"`@(_P#_`,(``O__A@`"X."&
M((L`#0$!`0($@("`#A$0$!.%``-98D.%``.,4MZ%``/@D)"0`$'0P```````
M``#_AP`!@,\`B""HS@`""``B&``(*BH8``@D!O@`"#PA#```````_P"&``R`
M8``/##`P#`PP`/^%``,\`/^%``/``/^%``$$1`#_``````````P#_`84`"L#_
M@,#@8"`@`/^'```'PAA"0``(&"(8`A!`,$A(2$V`0```\!(3.A``$<8J*BH0`
M#,`@("<````X*"!P((<(`0^'``&`_P#_`,<``S\@((8``N`<B``1!`4%`@("
M``"34E(B(B(``)N$20E(``"8)#P@))BR`(@(D0`Q`0$```$!`&"%A65EA85E
M,`P,,#`,##`B(CPB(B(\`$!,4DY24DX`!&25AH:59```@(0``8"A`(@@A0`+
M`0(*!!`0.$2"`0"`A@`"@$"0`(@0`H2$A@`"BG&&``(JRH8`#*"@`0$#`P4%
M,,```(0%"&`8!`3R"@D1_P#_`)H`_[^AOP``````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M``````````````````````````````````"<``(!`T(```````#__P(``(0@
M!N#@("`#`88`"/^`0"`0"`0$2O\``````````?B8`!<!`0```0$`986%966%
MA64P#`\P,`P,,$8``/\```````,``/Z%`H@``3^'`!3X"P0"`0```(#G@(`!
M@D0X(+]`@$(``````/\``(0``A#_AA!#`/\````````8!?P$`@(!`0"-B8G9
M<7$`P.$A(2(2#`08_P#_`*T``@$#0@```````/__"P```"`@(.#@("`@B``$
M`@$!`8@`'("`@$```'E$1'A$1```@("8I9VE```(",DJ#`S)`#<!`0```0$`
M986%966%A64P#`PP,`P,,`!$"D0H*1$1$0`-!,0DY`0D`("`F:6DI*0````$
M!*BH4)``0-#``````````/^'``'PAQ"8`(@0CP`!#(<``0B(``(X#X8(`F"`
M_P#_`*``_[^AOP``````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M`````````````````````````````````````````````````````+``B""0
M`(5`%7]`0$1X````_P``I9P```#_```JR4(```#_``````<```#_```'A`0;
M_`0$_P`],3TQ,`#_`+.VL['G`/\`O#`\L#P`A8`#_X"`0@``````_P``-0`!
M`0``_P``986%8&"````P#`PP/P```!````#_````Q````/\```"8````_P``
M`%````#_AP`$X"`@(*@`B!"8`(00!!\0$!"$`!3_`````@$!`/\``0$("`B(
MCX@("(0`!/\```"$"`3X"`@(_P#_`,(``O__A@`"X."&((L`#0$!`0($@("`
M#A$0$!.%``-98D.%``.,4MZ%``/@D)"0``$'0P````````H`_X<``8#/`(@@
MJ`"($)@`B!"(``L"#```GJ&AH0@("(D`!&"PD("("/\`_P"B`/^_H;\```$,
MAP`!"(@``C@/A@@"8(#_`/\`H`#_OZ&_````````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M````````````L`"(((4`)@$"'`0($"!`_P``$1$.``#_``!"0D$``/\``!!2
MC```_P``D)"04```_P``````#0```_P`("`P&`S_#!A%``````#_```($!`P
M8.#_P,!#``````#_``"%$`/P$!"(``2AH:&>A```)`2(B(AP``#`A(`#````
MB`C_`/\`R@`"(#^&``'@_P"P``,P(`"%(*@``F`PAA"8`(@0H`"("/\`_P"B
M`/^_H;\`_P`!`0@("(B/B`@(A``$_P```(0(!/@("`C_`/\`P@`"__^&``+@
MX(8@BP`-`0$!`@2`@(`.$1`0$X4``UEB0X4``XQ2WH4``^"0D)#_`.D`B""H
M`(@0F``*B!"@`(@(_P#_`/\`_P"$`(@@J`"($)@`B!"@`(@(_P#_`*(`_[^A
MOP``#0`` P``("`P&`S #!A%``````# ```($!`P8.# P,!#``````# ``"%
```

```
M$`/P$!"(``2AH:&>A``)`2(B(AP```#`A(`#````B`C_`/\`R@`"(#^&``'@
M_P"P``,P(`"%(*@``F`PAA"8`(@0H`"("/\`_P"B`/^_H;\`_P`!`0@("(B/
MB`@(A``$_P```(0(!/@("`C_`/\`P@`"__^&``+@X(8@BP`-`0$!`@2`@(`.
M$1`0$X4``UEB0X4``XQ2WH4``^"0D)#_`.D`B""H`(@0F`"($*``B`C_`/\`
M_P#S``$#AP(8_P``>V-[8V'_``!G;&9CSO\``'A@>&!XB("8`(@0B`"(`1G_
M```],3TQ,/\``+.VL['G_P``O#`\L#S`AT")``,?$!"$$Q@(_P``VQO;&P#_
M```[8S,;`/P$!,0$Q`3_`/\`D@#_OZ&_`(8``>#_`+```S`@`(4@J``"8#"&
M$)@`B!"@`(@(_P#_``*(`_[^AOP#_``$!"`@(B(^("`B$``3_````A`@$^`@(
M"/\`_P#"``+__X8``N#@AB"+``T!`0$"!("`@`X1$!`3A0`#66)#A0`#C%+>
MA0`#X)"0D/\`V0`#`@(#0@``````"@``_X40`P``_X4``X"`@)T`B!"(``,!
M`0%"`````````/^%"`,``/^%``-`0,"-``03$!`?A``$#@``_X0$!',``/^$
M``3$!`3\_P#_`/\`]P"($*@`B!"8`(@(H`"(!/\`_P"B`/^_H;\`!,0$Q`3_
M`/\`D@#_OZ&_`(8``>#_`+```S`@`(4@J``"8#"&$)@`B!"@`(@(_P#_``*(
M_[^AOP#_``$!"`@(B(^("`B$``3_````A`@$^`@("/\`_P#"``+__X8``N#@
MAB"+``T!`0$"!("`@`X1$!`3A0`#66)#A0`#C%+>A0`#X)"0D/\`Z0"($*@`
MB!"8`(@(H`"(!/\`_P#_`/\`A`"($*@`B!"8`(@(H`"(!/\`_P"B`/^_H;\`
M_X4``T!`P(T`!!,0$!^$``0.``#_A`0$<P``_X0`!,0$!/S_`/\`_P#W`(@0
MJ`"($)@`B!"@`(@$_P#_``*(`_[^AOP`$Q`3$!/\`_P"2`/^_H;\`A@`!X/\`
ML``#,"``A2"H``(P(8`0F`"(0J`"($/\`_P#B``#__^&``+@X(8@BP`-`0$!
M"(0`!/\```"$"`3X"`@(_P#_`,(``O__A@`"X."&(",`#0$!`0($@("`#A$0
M$!.%``-98D.%``.,4MZ%``/@D)"0_P"I`(@0J`"(")@`B!"@`(@$_P#_``*(
M`/@``P,$!(4``XX/#(4``XQ,#8N%``/,9>#_`*3`,`D`@8I$:`#W-R`@*0#`B
M(D5%A``$'`@``((X`@``?$!/P@(B`B!`<$!!-B$2!!9!N@*`(J`"8`(@@``F(
M@```````$!!P$!\!0$!\0$!\!H$A``$(-"(1`B!"(B!"@`(@@(_P#_``*(`
M`"!"@`(@$_P#_`/\`S@`"__^&``+@X(8@BP`-`0$!`@2`@(`.$1`0$X4``UEB
M0X4``XQ2WH4``^"0D)#_`*D`B!"H`(@(F`"($*``B`3_`/\`A`"($*@`B!"8
M`(@(H`"(!/\`_P#_`/\`U@""__^&``+@X(8@BP`-`0$!`@2`@(`.$1`0$X4`
M`UEB0X4``XQ2WH4``^"0D)#_`.D`B!"H`(@(F`"($*``B`3_`/\`A`"($*@`
MB!"8`(@(H`"(!/\`_P"B`/^_H;\`$,0$Q`3_`/\`D@#_OZ&_`(8``>#_`+``
M`S`@`(4@J``"8#"&$)@`B!"@`(@(_P#_``*(`_[^AOP#_``$!"`@(B(^("`B
M$``3_````A`@$^`@("/\`_P#"``+__X8``N#@AB"+``T!`0$"!("`@`X1$!`3
M```````
```

end

```
==============================================================================
```

LZW Compression

by Bill Lucier (Blucier@ersys.edmonton.ab.ca, b.lucier1 on Genie)

LZW is perhaps the most widely used form of data compression today. It is simple to implement and achieves very decent compression at a fairly quick pace. LZW is used in PKZIP (shrink),PKARC (crunch), gifs,V.42bis and unix's compress. This article will attempt to explain how the compression works with a short example and 6502 source code in Buddy format.

Originally named lz78, it was invented by Jacob Ziv and Abraham Lempel in 1978 , it was later modified by Terry Welch to its present format. The patent for the LZW compression method is presently held by Unisys.

LZW compresses data by taking phrases and compressing them into codes. The size of the codes could vary from 9 bits to 16 bits. Although for this implementation we will be using only 12 bits. As byte are read in from a file they are added to a dictionary. For a 12-bit implementation a dictionary will be 4k (2^12=4096) . Each entry in the dictionary requires five bytes, which will be built in the form of a tree. It is not a binary tree because each node may have more than two offsprings. In fact, because our dictionary can hold up to 4096 different codes it is possible to have one node with 3800 children nodes, although this is not likely to happen. The five bytes that make up our tree will be:

```
The parent code: Each node has one and only one parent node. When the parent
                 code is less then 255 it is the end of a phrase. The codes
                 0-255 do not actually exist in the tree. The following
                 values do not appear either as they have special meaning:

                 256 : End of Stream-This marks the end of one compressed file
                 257 : This tells the decompressor to increase the number
                       of bits its reading by one.
                 258 : Wipe out dictionary

The code value : This is the code that will be sent to the compressor.
The character  : The value contained at this node. It have a value of 0-255.
```

Initially we send out codes that are 9 bits long, this will cover the values 0-511. Once we have reached 511, we will need to increase the number of bits to write by 1. This will give room for code numbers 512-1023, or (2^10)-1. At this point we must ensure that the decompressor knows how bits to read in at once so a code number 257 is sent to indicate that the number of bits to be read is to be bumped up by one. The size of the dictionary is finite so at some point we do have to be concerned with what we will do when it does fill up. We could stop compiling new phrases and just compress with the ones that are already in the dictionary. This is not a very good choice, files tend to change frequently (eg. program files as they change from code to data) so sticking with the same dictionary will actually increase the size of the file or at best, give poor compression. Another choice is to wipe the dictionary out and start building new codes and phrases, or wipe out some of the dictionary leaving behind only the newer codes and phrases. For the sake of simplicity this program will just wipe out the dictionary when it becomes full.

To illustrate how LZW works a small phrase will be compressed : heher. To start the first two characters would be read in. The H would be treated as the parent code and E becomes the character code. By means of a hashing routine (the hashing routine will be explained more fully in the source code) the location where HE should be is located. Since we have just begun there will be nothing there,so the phrase will be added to the dictionary. The codes 0-258 are already taken so we start using 259 as our first code. The binary tree would look something like this:

```
          node # 72 - H
                 |
     node #3200 259 - E
```

The node # for E is an arbitrary one. The compressor may not choose that location, 3200 is used strictly for demonstration purposes. So at node #3200 the values would be:

Parent code - 72
code value - 259
character   - E

The node #72 is not actually used. As soon as a value less than 255 is found it is assumed to be the actual value. We can't compress this yet so the value 72 is sent to the output file(remember that it is sent in 9 bits). The E then becomes the parent code and a new character code (H) is read in. After again searching the dictionary the phrase EH is not found. It is added to the dictionary as code number 260. Then we send the E to the disk and H becomes the new parent code and the next E becomes the new character code. After searching the dictionary we find that we can compress HE into the code 259,we want to compress as much as possible into one code so we make 259 the parent code. There may be a longer string then HE that can be compressed. The R is read in as the new character code. The dictionary is searched for the a 259 followed a R, since it is not found it is added to the dictioary and it looks like this:

```
           node #72 - H             node #69 - E
                 |                        |
    node #3200  259 - E    node #1600    260 - H
                 |
    node #1262  261 - R
```

Then the value 259 is sent to the output file (to represent the HE) and since that is the EOF the R is sent as well,as well as a 256 to indicate the EOF has been reached.

Decompression is extremely simple. As long as the decompressor maintains the dictionary as the compressor did, there will be no problems,except for one problem that can be handled as an exceptional case. All of the little details of increasing the number of bits to read, and when to flush the dictionary are taken care of by the compressor. So if the dictionary was increased to 8k, the compressor would have to be set up to handle a larger dictionary, but the decompressor only does as the compressed file tells it to and will work with any size dictionary. The only problem would be that a larger dictionary will creep into the ram under the rom or possibly even use all available memory, but assuming that the ram is available the decompressor will not change. The decompressor would start out reading 9 bits at a time, and starts it free code at 259 as the compressor did. To use the above input from the compressor as an example, the output was:

72 - For the First H
69 - For the First E
259 - For the Compressed HE
82 - For the R
256 - Eof indicator

To begin decompressing, two values are needed. The H and E are read in, (note they will both be 9 bits long). As they are both below 256 they are at the end of the string and are sent straight to the output file. The first free code is 259 so that is the value assigned to the phrase HE. Note when decompressing there is no need for the hashing routine, the codes are the absolute locations in the dictionary (i.e. If the dictionary was considered to be an array then the entry number 259 would be dictionary[259]), because of this, the code value is no longer needed. So the decompressor would have an entry that looks like this:

Node # 259
Parent Code - H
Character   - E

The decompressor will read in the next value (259). Because the node number is at the end of the compressed string we will have to take the code value and place it on a stack, and take them off in a Last-in,First-out (LIFO) fashion. That is to say that the first character to go on the stack (in this case the E) will be the last to come off. The size of the stack is dependent on the size of the dictionary, so for this implementation we need a stack that is 4k long. After all the characters from the string have been placed on the stack they are taken off and sent to the outputfile.

There is one small error that is possible with LZW because of the way the compressor defines strings. Consider the compression dictionary that has the following in it:

node #	Code Value	Parent code	character
65	65	n/a	A
723	259	65	C
1262	260	259	U
2104	261	260	T
2506	262	261	E

Now if the compressor was to try to compress the string ACUTEACUTEA The compressor will find a match for the first five characters 'ACUTE' and will send a 262 to the output file. Then it will add the following entry to the dictionary:

3099   263   262   A

Now it will try to compress the remaining characters, and it finds that it can compress the entire string with the code 263, but notice that the middle A, the one that was just added onto the end of the string 'ACUTE' was never sent to the output file. The decompressor will not have the code 263 defined in it's dictionary. The last code it will have defined will be 262. This problem is easily remedied though, when the decompressor does not have a code defined, it takes the first letter from the last phrase defined and tacks it onto the end of the last phrase. IE It takes the first letter (the A) from the phrase and adds it on to the end as code #263.

This particular implementation is fairly slow because it reads a byte and then writes one, it could be made much faster with some buffering. It is also limited to compressing and decompressing one file at a time and has no error checking capabilities. It is meant strictly to teach LZW compression, not provide a full fledged compressor.

And now for the code:

```
SYS 4000       ; sys 999 on a 64
.DVO 9         ; or whatever drive used for output
.ORG 2500
.OBJ "LZW.ML"

TABLESIZE =5021

; THE TABLESIZE IS ACTUALLY 5021, ABOUT 20% LARGER THEN 4K. THIS GIVES
; THE HASHING ROUTINE SOME ROOM TO MOVE. IF THE TABLE WAS EXACTLY 4K
; THERE WOULD BE FREQUENT COLLISIONS WHERE DIFFERENT COMBINATIONS OF
; CHARACTERS WOULD HAVE THE SAME HASH ADDRESS. INCREASING THE TABLE SIZE
; REDUCES THE NUMBER OF COLLISIONS.

EOS =256           ; eos = End of stream This marks the end of file

FIRSTCODE =259
MAXCODE =4096

BUMPCODE =257      ; Whenever a 257 is encountered by the decompressor it
                   ; increases the number of bits it reads by 1
```

```
FLUSHCODE =258

TABLEBASE =14336  ; The location that the dictionary is located at

DECODESTACK =9300 ; The location of the 4k LIFO stack

; ORG = DECOMPRESS FILE
; ORG + 3 = COMPRESS FILE

JMP EXPANDFILE

;*******************************
; COMPRESSFILE
;*******************************

COMPRESSFILE JSR INITDIC ; EMPTY THE DICTIONARY
LDA #128
STA BITMASK
LDA #0
STA RACK
JSR GETCHAR      ; GET A CHAR FROM THE INPUT FILE
STA STRINGCODE   ; INITIALIZE THE STRINGCODE (PARENT CODE)
LDA #0
STA STRINGCODE+1
NEXTCHAR JSR GETCHAR
         STA CHARACTER
         JSR FINDNODE    ; FINDNODE CALCULATES THE HASHED LOCATION OF
         LDA ($FE),Y     ; THE STRINGCODE AND CHARACTER  IN THE DICT.
         INY             ; AND SETS $FE/$FF POINTING TO IT. IF THE ENTRY
         AND ($FE),Y     ; HAS TWO 255 IN IT THEN IT IS EMPTY AND SHOULD
         CMP #255        ; BE ADDED TO THE DICTIONARY.
         BEQ ADDTODICT
             LDA ($FE),Y     ; IT HAS A DEFINED PHRASE. STORE THE CODE VALUE IN
             STA STRINGCODE+1; THE PARENT CODE
             DEY
             LDA ($FE),Y
             STA STRINGCODE
         JMP EOF
         ADDTODICT LDY #0
         - LDA NEXTCODE,Y
           STA ($FE),Y
           INY
           CPY #5
         BNE -
         INC NEXTCODE             ; INCREASE THE NEXTCODE
         BNE +
             INC NEXTCODE+1
         + JSR OUTPUT
         LDA NEXTCODE+1          ; CHECK IF NEXTCODE=4096 IF SO THEN FLUSH THE
         CMP #>MAXCODE           ; DICTIONARY AND START ANEW
         BNE CHECKBUMP
         LDA NEXTCODE
         CMP #<MAXCODE
         BNE CHECKBUMP
         LDA #<FLUSHCODE        ; SEND THE FLUSH CODE TO THE COMPRESSED FILE SO
         STA STRINGCODE         ; THE DECOMPRESSOR WILL KNOW TO FLUSH THE
         LDA #>FLUSHCODE        ; DICTIONARY
         STA STRINGCODE+1
         JSR OUTPUT
         JSR INITDIC
         JMP CHECKEOF
         CHECKBUMP LDA NEXTBUMP+1
         CMP NEXTCODE+1         ; CHECKBUMP CHECK TO SEE IF THE NEXTCODE HAS
         BNE CHECKEOF        ; REACHED THE MAXIMUM VALUE FOR THE CURRENT
         LDA NEXTBUMP                    ; NUMBER OF BITS BEING OUTPUT.
         CMP NEXTCODE        ; FOR     X  BITS      NEXTCODE HAS Y PHRASES
         BNE CHECKEOF        ;        --------      -----------------------
         LDA #>BUMPCODE      ;           9                 511
         STA STRINGCODE+1    ;          10                1023
         LDA #<BUMPCODE      ;          11                2047
         STA STRINGCODE      ;          12                4095
         JSR OUTPUT
         INC CURRENTBITS
         ASL NEXTBUMP
         ROL NEXTBUMP+1
         CHECKEOF LDA #0
         STA STRINGCODE+1
         LDA CHARACTER
         STA STRINGCODE
         EOF LDA 144
```

```
         BNE DONE
JMP NEXTCHAR
DONE JSR OUTPUT
LDA #>EOS               ; SEND A 256 TO INDICATE EOF
STA STRINGCODE+1
LDA #<EOS
STA STRINGCODE
JSR OUTPUT
LDA BITMASK
BEQ +
    JSR $FFCC
    LDX #3
    JSR $FFC9
    LDA RACK              ; SEND WHAT BITS WEREN'T SEND WHEN OUTPUT
    JSR $FFD2
+ JSR $FFCC
LDA #3
JSR $FFC3
LDA #2
JMP $FFC3

;**********************************
; INITDIC
; INITIALIZES THE DICTIONARY, SETS
; THE NUMBER OF BITS TO 9
;**********************************

INITDIC LDA #9
STA CURRENTBITS
LDA #>FIRSTCODE
STA NEXTCODE+1
LDA #<FIRSTCODE
STA NEXTCODE
LDA #>512
STA NEXTBUMP+1
LDA #<512
STA NEXTBUMP
LDA #<TABLEBASE
STA $FE
LDA #>TABLEBASE
STA $FF
LDA #<TABLESIZE
STA $FC
LDA #>TABLESIZE
STA $FD
- LDY #0
  LDA #255      ; ALL THE CODE VALUES ARE INIT TO 255+256*255
  STA ($FE),Y   ; OR -1 IN TWO COMPLEMENT
  INY
  STA ($FE),Y
  CLC
  LDA #5        ; EACH ENTRY IN THE TABLE TAKES 5 BYTES
  ADC $FE
  STA $FE
  BCC +
      INC $FF
  + LDA $FC
  BNE +
      DEC $FD
  + DEC $FC
  LDA $FD
  ORA $FC
BNE -
RTS

;************************************
; GETCHAR
;************************************

GETCHAR JSR $FFCC
LDX #2
JSR $FFC6
JMP $FFCF

;************************************
; OUTPUT
;************************************

OUTPUT LDA #0      ; THE NUMBER OF BITS OUTPUT CAN BE OF A VARIABLE
STA MASK+1         ; LENGTH,SO THE BITS ARE ACCUMULATED TO A BYTE IS
LDA #1             ; FULL AND THEN IT IS SENT TO THE OUTPUT FILE
```

```
LDX CURRENTBITS
DEX
- ASL
  ROL MASK+1
  DEX
BNE -
STA MASK
MASKDONE LDA MASK
ORA MASK+1
BNE +
    RTS
+ LDA MASK
AND STRINGCODE
STA 3
LDA MASK+1
AND STRINGCODE+1
ORA 3
BEQ NOBITON
    LDA RACK
    ORA BITMASK
    STA RACK
NOBITON LSR BITMASK
LDA BITMASK
BNE +
    JSR $FFCC
    LDX #3
    JSR $FFC9
    LDA RACK
    JSR $FFD2
    LDA #0
    STA RACK
    LDA #128
    STA BITMASK
+ LSR MASK+1
ROR MASK
JMP MASKDONE

;******************************
; FINDNODE
; THIS SEARCHES THE DICTIONARY TILL IT FINDS A PARENT NODE THAT MATCHES
; THE STRINGCODE AND A CHILD NODE THAT MATCHES THE CHARACTER OR A EMPTY
; NODE.
;*******************************

; THE HASHING FUNCTION - THE HASHING FUNCTION IS NEEDED BECAUSE
; THERE ARE 4096 X 4096 (16 MILLION) DIFFERENT COMBINATIONS OF
; CHARACTER AND STRINGCODE. BY MULTIPLYING THE CHARACTER AND STRINGCODE
; IN A FORMULA WE CAN DEVELOP OF METHOD OF FINDING THEM IN THE
; DICTIONARY. IF THE STRINGCODE AND CHARACTER IN THE DICTIONARY
; DON'T MATCH THE ONES WE ARE LOOKING FOR WE CALCULATE AN OFFSET
; AND SEARCH THE DICTIONARY FOR THE RIGHT MATCH OR A EMPTY
; SPACE IS FOUND. IF AN EMPTY SPACE IS FOUND THEN THAT CHARACTER AND
; STRINGCODE COMBINATION IS NOT IN THE DICTIONARY

FINDNODE LDA #0
STA INDEX+1
LDA CHARACTER     ; HERE THE HASHING FUNCTION IS APPLIED TO THE
ASL               ; CHARACTER AND THE STRING CODE. FOR THOSE WHO
ROL INDEX+1       ; CARE THE HASHING FORMULA IS:
EOR STRINGCODE    ; (CHARACTER << 1) ^ STRINGCODE
STA INDEX         ; FIND NODE WILL LOOP TILL IT FINDS A NODE
LDA INDEX+1       ; THAT HAS A EMPTY NODE OR MATCHES THE CURRENT
EOR STRINGCODE+1  ; PARENT CODE AND CHARACTER
STA INDEX+1
ORA INDEX
BNE +
    LDX #1
    STX OFFSET
    DEX
    STX OFFSET+1
    JMP FOREVELOOP
+ SEC
LDA #<TABLESIZE
SBC INDEX
STA OFFSET
LDA #>TABLESIZE
SBC INDEX+1
STA OFFSET+1

FOREVELOOP JSR CALCULATE
           LDY #0
```

```
           LDA ($FE),Y
           INY
           AND ($FE),Y
           CMP #255
           BNE +
               LDY #0
               RTS
           + INY
           - LDA ($FE),Y
             CMP STRINGCODE-2,Y
             BNE +
             INY
             CPY #5
             BNE -
          LDY #0
          RTS
          + SEC
          LDA INDEX
          SBC OFFSET
          STA INDEX
          LDA INDEX+1
          SBC OFFSET+1
          STA INDEX+1
          AND #128
          BEQ FOREVELOOP
          CLC
          LDA #<TABLESIZE
          ADC INDEX
          STA INDEX
          LDA #>TABLESIZE
          ADC INDEX+1
          STA INDEX+1
JMP FOREVELOOP

;***************************
; CALCULATE
; TAKES THE VALUE IN INDEX AND CALCULATES ITS LOCATION IN THE DICTIONARY
;****************************

CALCULATE LDA INDEX
STA $FE
LDA INDEX+1
STA $FF
ASL $FE
ROL $FF
ASL $FE
ROL $FF
CLC
LDA INDEX
ADC $FE
STA $FE
LDA INDEX+1
ADC $FF
STA $FF
CLC
LDA #<TABLEBASE
ADC $FE
STA $FE
LDA #>TABLEBASE
ADC $FF
STA $FF
LDY #0
RTS

;******************************
; DECODESTRING
;******************************

DECODESTRING TYA   ; DECODESTRING PUTS THE STRING ON THE STACK
STA COUNT          ; IN A LIFO FASHION.
LDX #>DECODESTACK
CLC
ADC #<DECODESTACK
STA $FC
STX $FD
LDA #0
STA COUNT+1
- LDA INDEX+1
  BEQ +
  JSR CALCULATE
  LDY #4
```

```
  LDA ($FE),Y
  LDY #0
  STA ($FC),Y
  LDY #2
  LDA ($FE),Y
  STA INDEX
  INY
  LDA ($FE),Y
  STA INDEX+1
  JSR INFC
JMP -
+ LDY #0
LDA INDEX
STA ($FC),Y
INC COUNT
BNE +
    INC COUNT+1
+ RTS

;******************************
; INPUT
;******************************

INPUT LDA #0  ; THE INPUT ROUTINES IS USED BY THE DECOMPRESSOR
STA MASK+1    ; TO READ IN THE VARIABLE LENGTH CODES
STA RETURN
STA RETURN+1
LDA #1
LDX CURRENTBITS
DEX
- ASL
  ROL MASK+1
  DEX
BNE -
STA MASK
- LDA MASK
  ORA MASK+1
  BEQ INPUTDONE
  LDA BITMASK
  BPL +
      JSR GETCHAR
      STA RACK
  + LDA RACK
  AND BITMASK
  BEQ +
      LDA MASK
      ORA RETURN
      STA RETURN
      LDA MASK+1
      ORA RETURN+1
      STA RETURN+1
  + LSR MASK+1
  ROR MASK
  LSR BITMASK
  LDA BITMASK
  BNE +
      LDA #128
      STA BITMASK
+ JMP -
INPUTDONE RTS

;*******************************
; EXPANDFILE
; WHERE THE DECOMPRESSION IS DONE
;*******************************

EXPANDFILE LDA #0
STA RACK
LDA #128
STA BITMASK
START JSR INITDIC
JSR INPUT
LDA RETURN+1
STA OLDCODE+1       ; Save the first character in OLDCODE
LDA RETURN
STA CHARACTER
STA OLDCODE
CMP #<EOS
BNE +
    LDA RETURN+1    ; If return = EOS (256) then all done
    CMP #>EOS
```

```
    BNE +
    JMP CLOSE
+ JSR $FFCC
LDX #3
JSR $FFC9
LDA OLDCODE          ; Send oldcode to the output file
JSR $FFD2
NEXT JSR INPUT
     LDA RETURN
     STA NEWCODE
     LDA RETURN+1
     STA NEWCODE+1
     CMP #1          ; All of the special codes Flushcode,BumpCode & EOS
     BNE ++          ; Have 1 for a MSB.
         LDA NEWCODE
         CMP #<BUMPCODE
         BNE +
             INC CURRENTBITS
             JMP NEXT
         + CMP #<FLUSHCODE
         BEQ START
         CMP #<EOS
         BNE +
         JMP CLOSE
     + SEC           ; Here we compare the newcode just read in to the
     LDA NEWCODE     ; next code. If newcode is greater than it is a
     SBC NEXTCODE    ; ACUTEACUTEA situation and must be handle differently.
     STA 3
     LDA NEWCODE+1
     SBC NEXTCODE+1
     ORA 3
     BCC +
         LDA CHARACTER
         STA DECODESTACK
         LDA OLDCODE
         STA INDEX
         LDA OLDCODE+1
         STA INDEX+1
         LDY #1
         BNE ++
     + LDA NEWCODE  ; Point index to newcode spot in the dictionary
     STA INDEX       ; So DECODESTRING has a place to start
     LDA NEWCODE+1
     STA INDEX+1
     LDY #0
     + JSR DECODESTRING
     LDY #0
     LDA ($FC),Y
     STA CHARACTER
     INC $FC
     BNE +
         INC $FD
     + JSR $FFCC
     LDX #3
     JSR $FFC9
     L1 LDA COUNT+1  ; Count contains the number of characters on the stack
        ORA COUNT
        BEQ +
        JSR DECFC
        LDY #0
        LDA ($FC),Y
        JSR $FFD2
     JMP L1
     + LDA NEXTCODE  ; Calculate the spot in the dictionary for the string
     STA INDEX        ; that was just entered.
     LDA NEXTCODE+1
     STA INDEX+1
     JSR CALCULATE
     LDY #2           ; The last character read in is tacked onto the end
     LDA OLDCODE      ; of the string that was just taken off the stack
     STA ($FE),Y      ; The nextcode is then incremented to prepare for the
     INY              ; next entry.
     LDA OLDCODE+1
     STA ($FE),Y
     INY
     LDA CHARACTER
     STA ($FE),Y
     INC NEXTCODE
     BNE +
         INC NEXTCODE+1
     + LDA NEWCODE
```

```
     STA OLDCODE
     LDA NEWCODE+1
     STA OLDCODE+1
JMP NEXT
CLOSE JSR $FFCC
LDA #2
JSR $FFC3
LDA #3
JMP $FFC3

DECFC LDA $FC
BNE +
    DEC $FD
+ DEC $FC
LDA COUNT
BNE +
    DEC COUNT+1
+ DEC COUNT
RTS
INFC INC $FC
BNE +
    INC $FD
+ INC COUNT
BNE +
    INC COUNT+1
+ RTS

NEXTCODE .WOR 0
STRINGCODE .WOR 0
CHARACTER .BYT 0
NEXTBUMP .WOR 0
CURRENTBITS .BYT 0
RACK .BYT 0
BITMASK .BYT 0
MASK .WOR 0
INDEX .WOR 0
OFFSET .WOR 0
RETURN .WOR 0
COUNT .WOR 0
NEWCODE .WOR 0
OLDCODE .WOR 0
TEST .BYT 0
```

TO DRIVE THE ML I WROTE THIS SMALL BASIC PROGRAM. NOTE THAT CHANNEL TWO IS ALWAYS THE INPUT AND CHANNEL THREE IS ALWAYS THE OUTPUT. EX AND CO MAY BE CHANGED TO SUIT WHATEVER LOCATIONS THE PROGRAM IS REASSEMBLED AT.

```
1 IFA=.THENA=1:LOAD"LZW.ML",PEEK(186),1
10 EX=2500:CO=2503
15 PRINT"[E]XPAND OR [C]OMPRESS?"
20 GETA$:IFA$<>"C"ANDA$<>"E"THEN20
30 INPUT"NAME OF INPUT FILE";FI$:IFLEN(FI$)=.THEN30
40 INPUT"NAME OF OUTPUT FILE";FO$:IFLEN(FO$)=.THEN40
50 OPEN2,9,2,FI$+",P,R":OPEN3,9,3,FO$+",P,W"
60 IFA$="E"THENSYSEX
70 IFA$="C"THENSYSCO
80 END
```

For those interested in learning more about data compression/decompression I recommend the book 'The Data Compression Book' written by Mark Nelson. I learned a great deal from reading this book. It explains all of the major data compression methods. (huffman coding, dictionary type compression such as LZW, arithmatic coding, speech compression and lossy graphics compression)

Questions or comments are welcome, they may be directed to me at :

Internet   : Blucier@ersys.edmonton.ab.ca
Genie   : b.lucier1

```
-------------------------------------------------------------------------------

begin 644 lzw.ml
MQ`E,N`P@GPJI@(WT#:D`C?,-(.L*C>T-J0"-[@T@ZPJ-[PT@6`NQ_L@Q_LG_
M\`ZQ_HWN#8BQ_HWM#4QH"J``N>L-D?[(P`70]N[K#=`#[NP-(/8*K>P-R1#0
M&JWK#<D`T!.I`HWM#:D!C>X-(/8*()\*3%T*K?$-S>P-T!ZM\`W-ZPW0%JD!
MC>X-J0&-[0T@]@KN\@T.\`TN\0VI`(WN#:WO#8WM#:60T`-,WPD@]@JI`8WN
M#:D`C>T-(/8*K?0-\`X@S/^B`R#)_ZWS#2#2_R#,_ZD#(,/_J0),P_^I"8WR
M#:D!C>P-J0.-ZPVI`HWQ#:D`C?`-J0"%_JDXA?^IG87\J1.%_:``J?^1_LB1
M_ABI!67^A?Z0`N;_I?S0`L;]QORE_07\T-Y@(,S_H@(@QO],S_^I`(WV#:D!
MKO(-R@HN]@W*T/F-]0VM]0TM[0V%`ZWV#2WN#04#\`FM\PT-
```

```
M]`V-\PU.]`VM]`W0&"#,_Z(#(,G_K?,-(-+_J0"-\PVI@(WT#4[V#6[U#4P+
M"ZD`C?@-K>\-"B[X#4WM#8WW#:WX#4WN#8WX#=`1K?<-T`RB`8[Y#<J.^@U,
MEPLXJ9WM]PV-^0VI$^WX#8WZ#2#C"Z``L?[(,?[)_]`#H`!@R+'^V>L-T`C(
MP`70]*``8#BM]PWM^0V-]PVM^`WM^@V-^`TI@/`1&*F=;?<-C?<-J1-M^`V-
M^`U,EPNM]PV%_JWX#87_!OXF_P;^)O\8K?<-9?Z%_JWX#67_A?\8J0!E_H7^
MJ3AE_X7_H`!@F(W]#:(D&&E4A?R&_:D`C?X-K?@-\!X@XPN@!+'^H`"1_*``"
ML?Z-]PW(L?Z-^`T@W`U,)@R@`*WW#9'\[OT-T`/N_@U@J0"-]@V-^PV-_`VI
M`:[R#<H*+O8-RM#YC?4-K?4-#?8-\#NM]`T0!B#K"HWS#:WS#2WT#?`2K?4-
M#?L-C?L-K?8-#?P-C?P-3O8-;O4-3O0-K?0-T`6I@(WT#4QT#&"I`(WS#:F`
MC?0-()\*(%D,K?P-C0(.K?L-C>\-C0$.R0#0"JW\#<D!T`-,NPT@S/^B`R#)
M_ZT!#B#2_R!9#*W[#8W_#:W\#8T`#LD!T!BM_PW)`=`&[O(-3/,,R0+PJ\D`
MT`-,NPTXK?\-[>L-A0.M``[M[`T%`Y`6K>\-C50DK0$.C?<-K0(.C?@-H`'0
M#JW_#8WW#:T`#HWX#:``(!0,H`"Q_(WO#>;\T`+F_2#,_Z(#(,G_K?X-#?T-
M\`T@R`V@`+'\(-+_3&T-K>L-C?<-K>P-C?@-(.,+H`*M`0Z1_LBM`@Z1_LBM
M[PV1_N[K#=`#[NP-K?\-C0$.K0`.C0(.3/,,(,S_J0(@P_^I`TS#_Z7\T`+&
M_<;\K?T-T`/._@W._0U@YOS0`N;][OT-T`/N_@U@````````````````````
*````````````````
`

end

crc32 for lzw.ml = 2460116527

begin 644 lzw.bas
M`0@<"`$`BT&R+J=!LC$ZDR),6E<N34PB+#DL,0`P"`H`15BR,C4P,#I#3[(R
M-3`S`%`(#P"9(I,219)84$%.1"!/4B`20Y)/35!215-3/R(`;`@4`*%!)#J+
M022SL2)#(J]!)+.Q(D4BIS(P`)<('@"%(DY!344@3T8@24Y0550@1DE,12([
M1DDD.HO#*$9))"FR+J<S,`##""@`A2).04U%($]&($]55%!55"!&24Q%(CM&
M3R0ZB\,H1D\D*;(NIS0P`.L(,@"?,BPY+#(L1DDDJB(L4RQ2(CJ?,RPY+#,L
M1D\DJB(L4"Q7(@#["#P`BT$DLB)%(J>>15@`"PE&`(M!)+(B0R*GGD-/````
`

end

crc32 for lzw.bas = 100674089
```

THREE-KEY ROLLOVER for the C-128 and C-64.

by Craig Bruce <csbruce@neumann.uwaterloo.ca>

1. INTRODUCTION

This article examines a three-key rollover mechanism for the keyboards of the C-128 and C-64 and presents Kernal-wedge implementations for both machines. Webster's doesn't seem to know, so I'll tell you that this means that the machine will act sensibly if you are holding down one key and then press another without releasing the first (or even press a third key while holding down two others). This is useful to fast touch typers. In fact, fast typing without rollover can be quite annoying; you get a lot of missing letters.

Another annoying property of the kernel keyscanning is joystick interference. If you move the joystick plugged into port #1, you will notice that some junk keystrokes result. The keyscanners here eliminate this problem by simply checking if the joystick is pressed and ignoring the keyboard if it is.

The reason that a 3-key rollover is implemented instead of the more general N-key rollover is that scanning the keyboard becomes more and more unreliable as more keys are held down. Key "shaddows" begin to appear to make it look like you are holding down a certain key when you really are not. So, by limiting the number of keys scanned to 3, some of this can be avoided. You will get strange results if you hold down more than three keys at a time, and even sometimes when holding down 3 or less. The "shift" keys (Shift, Commodore, Control, Alternate, and CapsLock) don't count in the 3 keys of rollover, but they do make the keyboard harder to read correctly. Fortunately, three keys will allow you to type words like "AND" and "THE" without releasing any keys.

2. USER GUIDE

Using these utilities is really easy - you just type away like normal. To install the C-128 version, enter:

```
BOOT "KEYSCAN128"
```

and you're in business. The program will display "Keyscan128 installed" and go to work. The program loads into memory at addresses $1500-$17BA (5376-6074 decimal), so you'll want to watch out for conflicts with other utilities. This program also takes over the IRQ vector and the BASIC restart vector ($A00). The program will survive a RUN/STOP+RESTORE. To uninstall this program, you must reset the machine (or poke the kernel values back into the vectors); it does not uninstall itself.

Loading the C-64 version is a bit trickier, so a small BASIC loader program is provided. LOAD and RUN the "KEYSCAN64.BOOT" program. It will load the "KEYSCAN64" program into memory at addresses $C500-$C77E (50432-51070 decimal) and execute it (with a SYS 50432). To uninstall the program, enter SYS 50435. The program takes over the IRQ and NMI vectors and only gives them back to the kernel upon uninstallation. The program will survive a RUN/STOP+RESTORE.

Something that you may or may not know about the C-64 is that its keys can be made to repeat by poking to address 650 decimal. POKE650,128 will enable the repeating of all keys. POKE650,0 will enable only the repeating of the SPACE, DELETE, and CURSOR keys. POKE650,64 will disable the repeating of all keys. An unusual side effect of changing this to either full repeat or no repeat is that holding down SHIFT+COMMODORE (character set shift) will repeat rapidly.

To see the rollover in action, hold down the "J" key for a while, and then press "K" without releasing "J". "K" will come out as expected, as it would with the kernal. Now, release the "J" key. If you are on a C-128, you will notice that the "K" key will now stop repeating (this is actually an important feature - it avoids problems if you don't release all of the keys you are holding down, at once). Now, press and hold the "J" key again without releasing the "K". "J" will now appear. It wouldn't using the Kernal key scanner. You can also try this with 3-key combinations. There will be some combinations that cause problems; more on this below.

Also, take a spaz on the joystick plugged into port #1 and observe that no garbage gets typed in. This was an annoying problem with the kernel of both the 64 and 128 and has lead many different games to picking between joystick #1 and #2 as the primary controller. The joystick in port #2 is not a problem to either Keyscan-128/64 or the Kernal.

3. KEYBOARD SCANNING

The Kernal scans the keyboard sixty times a second to see what keys you are holding down. Because of hardware peculiarities, there are multiple scanning techniques that will give different results.

3.1. SCANNING EXAMPLE

An example program is included to demonstrate different keyboard scanning techniques possible. To run it from a C-128 in 40-column (slow) mode, enter:

```
BOOT "KEYSHOW"
```

On a C-64, you must:

```
LOAD "KEYSHOW",8,1
```

and then:

```
SYS 4864
```

The same program works on both machines. Four maps of the keyscanning matrix will be displayed on the 40-column screen, as scanned by different techniques. The leftmost one is scanned from top to bottom "quickly". The second from the left scans from bottom to top "quickly". The third from the left scans the keyboard sideways, and the rightmost matrix scans the keys from top to bottom "slowly".

The mapping of keyscan matrix positions to keys is as follows:

```
ROWS: \             COLUMNS:    peek($DC01)
poke   \
$DC00   \   128      64      32      16      8       4       2       1
         +-------+-------+-------+-------+-------+-------+-------+-------+
  255-1  | DOWN  |   F5  |   F3  |   F1  |   F7  | RIGHT | RETURN| DELETE|
         +-------+-------+-------+-------+-------+-------+-------+-------+
  255-2  |LEFT-SH|   E   |   S   |   Z   |   4   |   A   |   W   |   3   |
         +-------+-------+-------+-------+-------+-------+-------+-------+
  255-4  |   X   |   T   |   F   |   C   |   6   |   D   |   R   |   5   |
         +-------+-------+-------+-------+-------+-------+-------+-------+
  255-8  |   V   |   U   |   H   |   B   |   8   |   G   |   Y   |   7   |
         +-------+-------+-------+-------+-------+-------+-------+-------+
 255-16  |   N   |   O   |   K   |   M   |   0   |   J   |   I   |   9   |
         +-------+-------+-------+-------+-------+-------+-------+-------+
 255-32  |   ,   |   @   |   :   |   .   |   -   |   L   |   P   |   +   |
         +-------+-------+-------+-------+-------+-------+-------+-------+
 255-64  |   /   |   ^   |   =   |RGHT-SH| HOME  |   ;   |   *   |   \   |
         +-------+-------+-------+-------+-------+-------+-------+-------+
255-128  | STOP  |   Q   |COMMODR| SPACE |   2   |CONTROL|   _   |   1   |
         +-------+-------+-------+-------+-------+-------+-------+-------+
```

The following table contains the additional keys which must be scanned on the

C128 (but which are not displayed by the example scanning program).

ROWS: poke $D02F	COLUMNS: peek($DC01)							
	128	64	32	16	8	4	2	1
255-1	1	7	4	2	TAB	5	8	HELP
255-2	3	9	6	ENTER	LF	-	+	ESC
255-4	NO-SCRL	RIGHT	LEFT	DOWN	UP	.	0	ALT

These tables are presented on page 642 of the Commodore 128 Programmer's Reference Guide. The scan codes that are stored in location 212 on the C128 and location 197 on the C64 are calculated based on the above tables. The entry in the "1" bit position of the first line of the first table (DELETE) has a scan code of 0, the "2" entry (RETURN) has a scan code of 1, etc., the entry on the second scan line in the "1" position ("3") has a scan code of 8, etc., all the way down to code 63. The scan codes for the 128 go all the way to 87, continuing in the second table like the first.

You will notice some strange effects of the different scanning techniques when you hold down multiple keys. More on this below. Also try pushing joystick #1 around.

3.2. SCANNING HARDWARE

To scan the 128 keyboard, you must poke a value into $DC00 (CIA#1 port A) and $D02F (VIC chip keyboard select port) to select the row to be scanned. The Data Direction Register for this port will be set to all outputs by the Kernal, so you don't have to worry about it. Each bit of $DC00 and the three least significant bits of $D02F are used for selecting rows. A "0" bit means that a row IS selected, and a "1" means that a row IS NOT selected. The poke value to use for selecting among the various rows are given in the two tables in the previous section.

Using one bit per row allows you to select multiple rows at the same time. It can be useful to select all rows at one time or no rows. To read the row that has been selected, simply peek at location $DC01 (CIA#1 port B). Each bit will tell you whether the corresponding key is currently being held down or not. Again, we have reverse logic; a "0" means that the key is being held down, and a "1" means that the key is not held down. The bit values corresponding to the keys are given as the column headings in the tables in the previous section. Since there is no such thing as a perfect mechanical switch, it is recommended that you "debounce" each key row read in the following way:

```
again:
   lda $dc01
   cmp $dc01
   bne again
```

So, to scan the entire keyboard, you simply select each scan row in some order, and read and remember the keys held down on the row. As it turns out, you have to be a bit careful of exactly how you "select" a row. Also, there is a shortcut that you can take. In order to find out if any key is being held down in one operation, all you have to do is select all rows at once and see if there are any "0" bits in the read value. If so, there is a key being held down somewhere; if not, then there is no key being held down, so you don't have to bother scanning the entire keyboard. This will reduce our keyscanning significantly, which is important, since the keyboard will be scanned every 1/60 of a second.

As mentioned above, joystick #1 will interfere with the Kernal reading the keyboard. This is because the read value of joystick #1 is wired into CIA#1 port A, the same place that the keyboard read is wired in. So, whenever a switch in the joystick is pushed, the corresponding bit of the keyboard scan register will be forced to "0", regardless of which keys are pressed and regardless of which scan row is selected. There's the catch. If we were to un-select all scan rows and still notice "0"s in the keyboard read register, then we would know that the joystick was being pushed and would interfere with our keyboard scanning, so we could abort keyboard scanning and handle this case as if no keys were being held down.

It still would be possible but unlikely that the user could push the joystick in the middle of us scanning the keyboard and screw up our results, so to defend against this, we check for the joystick being pushed both before and after scanning the keyboard. If we find that the joystick is pushed at either of these times, then we throw out the results and assume that no keys are held down. This way, the only way that a user could screw up the scanning is if he/she/it were to press a switch after we begin scanning and release it before we finish scanning. Not bloody likely for a human.

You get the same deal for keyboard scanning on the 64, except you only need to use $DC00 for selecting the scan rows. Also note that you will not be able to play with keyboard scanning from BASIC because of the interrupt reading of the keyboard. You must make sure that interrupts are disabled when playing with the keyboard hardware, or interrupt scanning can come along at any time and change all of the register settings.

3.3. SCANNING SOURCE CODE

The four keyboard scanning techniques of the example program are presented below. The declarations required for all of them are:

```
pa = $dc00         ;row select
pb = $dc01         ;column read
ddra = $dc02       ;ddr for row select
ddrb = $dc03       ;ddr for column read
scanTable .buf 8  ;storage for scan
mask = $03         ;work location
```

The code is as follows, in Buddy format. Each routine scans the keyboard and stores the results in the "scanTable" table.

```
------------------+------------------+------------------+------------------+
 Row forward fast | Row backward fast| Column right     | Row forward slow
------------------+------------------+------------------+------------------+
  sei             |  sei             |  sei             |  sei
  ldx #0          |  ldx #7          |  lda #$00        |  ldx #0
  lda #$fe        |  lda #$7f        |  sta ddra        |  lda #$fe
  sta pa          |  sta pa          |  lda #$ff        |  sta mask
  nextRow = *     |  nextRow = *     |  sta ddrb        |  nextRow = *
- lda pb          |- lda pb          |  ldy #7          |  lda mask
  cmp pb          |  cmp pb          |  lda #$7f        |  sta pa
  bne -           |  bne -           |  sta mask        |- lda pb
  eor #$ff        |  eor #$ff        |  nextCol = *     |  cmp pb
  sta scanTable,x |  sta scanTable,x |  lda mask        |  bne -
  sec             |  sec             |  sta pb          |  eor #$ff
  rol pa          |  ror pa          |- lda pa          |  sta scanTable,x
  inx             |  dex             |  cmp pa          |  sec
  cpx #8          |  bpl nextRow     |  bne -           |  rol mask
  bcc nextRow     |  cli             |  ldx #$ff        |  inx
  cli             |  rts             |  stx pb          |  cpx #8
  rts             |                  |  eor #$ff        |  bcc nextRow
------------------+------------------+  ldx #7          |  cli
                                     |- asl             |  rts
                                     |  rol scanTable,x +------------------+
                                     |  dex             |
                                     |  bpl -           |
                                     |  sec             |
                                     |  ror mask        |
                                     |  dey             |
                                     |  bpl nextCol     |
                                     |  lda #$ff        |
                                     |  sta ddra        |
                                     |  lda #$00        |
                                     |  sta ddrb        |
                                     |  cli             |
                                     |  rts             |
                                     +------------------+
```

The forward "quick" scanning stores the scan row selection mask into the row selection register and shifts the "0" bit one position "forward" for each row, directly, using a "rol $dc00" instruction. This would probably be the obvious solution to an optimizing assembler programmer. However, for some reason not quite understood by this author, there are "shadowing" problems with this approach. If you were to hold down the two keys "H" and "K" at the same time, you would notice that these two keys are on the same column of two successive rows. If you hold them both down, you will see the two positions become active, but so will the same column of all successive rows after the "H" and "K", even though these other keys are not actually held down. You will get an inaccurate reading if bad keys are held down simultaneously. You will notice the use of the term "active" above. This is because although the hardware returns a "0" for active, the routine converts that into a "1" for easier processing later. I am not sure if everyone will get this same result, but if your keyboard is wired the same as mine, you will.

The backward "quick" scanning operates quite similarly to the forward scanning, except for the direction of the scan and the direction of the "shadow"; the shadow goes upwards. You might think that ANDing together the results of the forward and backward scan together would eliminate the shadow, but this will not work since any rows between the rows containing the two keys held down will be incorrectly read as being active.

The columnwise right scanning is the most complicated because the rows must be converted into columns, to allow the scan matrix to be interpreted as before.

Also, the Data Direction Registers have to be changed. You might think that combinging row-wise scanning with columnwise scanning would give better results, and it probably would, if it weren't for a bizarre hardware problem. If you hold down two or more keys on the same scan row, say "W" and "E", some of the keys will flicker or disappear, giving an inaccurate reading.

The forward "slow" scanning is the best of the bunch. Incidentally, it is what the Kernal uses (as near as I can figure - their code is extremely convoluted). This technique is the same as the forward "quick scan," except that the row selection mask is shifted in a working storage location and poked into the CIA register, rather than being shifted in place. I don't know why this makes a difference, but it does. There is still a problem with this technique, but this problem occurs with all techniques. If you hold down three keys that form three "corners" of a rectangle in the scanning matrix, then the missing corner will be read as being held down also. For example, if you hold down "C", "N", and "M", then the keyboard hardware will also think that you are holding down the "X" key. This is why this article implements a "three-key" rollover rather than an "N-key" rollover. Many three-key combinations will still be interpreted correctly. Note, however, that shift keys such as SHIFT or CONTROL will add one more key to the hardware scanning (but will not be counted in the three-key rollover), making inaccurate results more likely if you are holding down multiple other keys at the same time.

4. THE C-128 KEYSCANNER

This section gives the source code for the C-128 implementation of the three-key rollover. The forward "slow" key matrix scanning technique is used, extended to work with the extra keys of the 128. It was a bit of a pain wedging into the Kernal, since there is not a convenient indirect JMP into scanning the keyboard, like there are for decoding and buffering pressed keys. A rather lengthy IRQ "preamble" had to be copied from the ROM, up to the point where it JSRs to the keyscanning routine. This code in included in the form of a ".byte" table, to spare you the details.

Before scanning the keyboard, we check to see if joystick #1 is pushed and if a key is actually pressed. If not, we abort scanning and JMP to the key repeat handling in the ROM. If a key is held down, we scan the keyboard and then examine the result. First we check for the shift keys (SHIFT, COMMODORE, CONTROL, ALT, and CAPS LOCK), put them into location $D3 (shift flags) in bit postitions 1, 2, 4, 8, and 16, respectively, and remove them from the scan matrix. The CAPS LOCK key is not on the main key matrix; it is read from the processor I/O port. This is good, because otherwise we could not abort scanning if it were the only key held down.

Then we scan the keymatrix for the first three keys that are being held down, or as many as are held down if less than three. We store the scan codes of these keys into a 3-element array. We also retain a copy of the 3-element array from the previous scan and we check for different keys being in the two arrays. If the old array contains a key that is not present in the new array, then the use has released a key, so we set a flag to inhibit interpretation of keys and pretend that no keys are held down. This is to eliminate undesirable effects of having other keys held down repeat if you release the most recently pushed key first. PC keyboards do this. This inhibiting will be ignored if new keys are discovered in the next step.

If there are keys in the new array that are not in the old, then the user has just pressed a new key, so that new key goes to the head of the old array and we stop comparing the arrays there. The key in the first position of the old array is poked into the Kernal "key held down" location for the Kernal to interpret later. If more than one new key is discovered at the same time, then each of the new keys will be picked up on successive keyboard scans and will be interpreted as just being pushed. So, if you press the "A", "N", and "D" keys all at the same time, some permutation of all three of these keys will appear on the screen.

When we are done interpreting the keys, we check the joystick once more and if it is still inactive, we present the most recently pushed down key to the Kernal and JMP into the ROM keyboard decoding routine.

Unlike in previous issues, this source code is here in literal form; just extract everything between the "-----=-----"s to nab the source for yourself. The source is in Buddy assembler format.

```
-----=-----
;3-Key Rollover-128 by Craig Bruce 18-Jun-93 for C= Hacking magazine

.org $1500
.obj "@0:keyscan128"

scanrows = 11
rollover = 3
```

```
pa = $dc00
pb = $dc01
pk = $d02f

jmp initialInstall

;ugly IRQ patch code.

irq = *  ;$1503
   .byte $d8,$20,$0a,$15,$4c,$69,$fa,$38,$ad,$19,$d0,$29,$01,$f0,$07,$8d
   .byte $19,$d0,$a5,$d8,$c9,$ff,$f0,$6f,$2c,$11,$d0,$30,$04,$29,$40,$d0
   .byte $31,$38,$a5,$d8,$f0,$2c,$24,$d8,$50,$06,$ad,$34,$0a,$8d,$12,$d0
   .byte $a5,$01,$29,$fd,$09,$04,$48,$ad,$2d,$0a,$48,$ad,$11,$d0,$29,$7f
   .byte $09,$20,$a8,$ad,$16,$d0,$24,$d8,$30,$03,$29,$ef,$2c,$09,$10,$aa
   .byte $d0,$28,$a9,$ff,$8d,$12,$d0,$a5,$01,$09,$02,$29,$fb,$05,$d9,$48
   .byte $ad,$2c,$0a,$48,$ad,$11,$d0,$29,$5f,$a8,$ad,$16,$d0,$29,$ef,$aa
   .byte $b0,$08,$a2,$07,$ca,$d0,$fd,$ea,$ea,$aa,$68,$8d,$18,$d0,$68,$85
   .byte $01,$8c,$11,$d0,$8e,$16,$d0,$b0,$13,$ad,$30,$d0,$29,$01,$f0,$0c
   .byte $a5,$d8,$29,$40,$f0,$06,$ad,$11,$d0,$10,$01,$38,$58,$90,$07,$20
   .byte $aa,$15,$20,$e7,$c6,$38,$60

;keyscanning entry point

main = *
   lda #0               ;check if any keys are held down
   sta pa
   sta pk
-  lda pb
   cmp pb
   bne -
   cmp #$ff
   beq noKeyPressed     ;if not, then don't scan keyboard, goto Kernal

   jsr checkJoystick    ;if so, make sure joystick not pressed
   bcc joystickPressed
   jsr keyscan          ;scan the keyboard and store results
   jsr checkJoystick    ;make sure joystick not pressed again
   bcc joystickPressed
   jsr shiftdecode      ;decode the shift keys
   jsr keydecode        ;decode the first 3 regular keys held down
   jsr keyorder         ;see which new keys pressed, old keys released, and
                        ;  determine which key to present to the Kernal
   lda $033e            ;set up for and dispatch to Kernal
   sta $cc
   lda $033f
   sta $cd
   ldx #$ff
   bit ignoreKeys
   bmi ++
   lda prevKeys+0
   cmp #$ff
   bne +
   lda $d3
   beq ++
   lda #88
+  sta $d4
   tay
   jmp ($033a)

   noKeyPressed = *     ;no keys pressed; select default scan row
   lda #$7f
   sta pa
   lda #$ff
   sta pk

   joystickPressed = *
   lda #$ff             ;record that no keys are down in old 3-key array
   ldx #rollover-1
-  sta prevKeys,x
   dex
   bpl -
   jsr scanCaps         ;scan the CAPS LOCK key
   ldx #$ff
   lda #0
   sta ignoreKeys

+  lda #88              ;present "no key held" to Kernal
   sta $d4
   tay
   jmp $c697
```

```
initialInstall = *       ;install wedge: set restore vector, print message
   jsr install
   lda #<reinstall
   ldy #>reinstall
   sta $0a00
   sty $0a01
   ldx #0
-  lda installMsg,x
   beq +
   jsr $ffd2
   inx
   bne -
+  rts

   installMsg = *
   .byte 13
   .asc "keyscan128 installed"
   .byte 0

reinstall = *            ;re-install wedge after a RUN/STOP+RESTORE
   jsr install
   jmp $4003

install = *              ;guts of installation: set IRQ vector to patch code
   sei                   ;  and initialize scanning variables
   lda #<irq
   ldy #>irq
   sta $0314
   sty $0315
   cli
   ldx #rollover-1
   lda #$ff
-  sta prevKeys,x
   dex
   bpl -
   lda #0
   sta ignoreKeys
   rts

mask = $cc

keyscan = *              ;scan the (extended) keyboard using the forward
   ldx #$ff              ;  row-wise "slow" technique
   ldy #$ff
   lda #$fe
   sta mask+0
   lda #$ff
   sta mask+1
   jmp +
   nextRow = *
-  lda pb
   cmp pb
   bne -
   sty pa
   sty pk
   eor #$ff
   sta scanTable,x
   sec
   rol mask+0
   rol mask+1
+  lda mask+0
   sta pa
   lda mask+1
   sta pk
   inx
   cpx #scanrows
   bcc nextRow
   rts

shiftValue = $d3

shiftRows .byte $01,$06,$07,$07,$0a
shiftBits .byte $80,$10,$20,$04,$01
shiftMask .byte $01,$01,$02,$04,$08

shiftdecode = *          ;see which "shift" keys are held down, put them into
   jsr scanCaps          ;  proper positions in $D3 (shift flags), and remove
   ldy #4                ;  them from the scan matrix
-  ldx shiftRows,y
   lda scanTable,x
```

```
   and shiftBits,y
   beq +
   lda shiftMask,y
   ora shiftValue
   sta shiftValue
   lda shiftBits,y
   eor #$ff
   and scanTable,x
   sta scanTable,x
+  dey
   bpl -
   rts

scanCaps = *             ;scan the CAPS LOCK key from the processor I/O port
-  lda $1
   cmp $1
   bne -
   eor #$ff
   and #$40
   lsr
   lsr
   sta shiftValue
   rts

newpos = $cc
keycode = $d4
xsave = $cd

keydecode = *            ;get the scan codes of the first three keys held down
   ldx #rollover-1       ;initialize: $ff means no key held
   lda #$ff
-  sta newKeys,x
   dex
   bpl -
   ldy #0
   sty newpos
   ldx #0
   stx keycode

   decodeNextRow = *     ;decode a row, incrementing the current scan code
   lda scanTable,x
   beq decodeContinue
                         ;at this point, we know that the row has a key held
   ldy keycode
-  lsr
   bcc ++
   pha                   ;here we know which key it is, so store its scan code,
   stx xsave             ;  up to 3 keys
   ldx newpos
   cpx #rollover
   bcs +
   tya
   sta newKeys,x
   inc newpos
+  ldx xsave
   pla
+  iny
   cmp #$00
   bne -

   decodeContinue = *
   clc
   lda keycode
   adc #8
   sta keycode
   inx
   cpx #scanrows
   bcc decodeNextRow
   rts

;keyorder: determine what key to present to the Kernal as being logically the
;only one pressed, based on which keys previously held have been released and
;which new keys have just been pressed

keyorder = *
   ;** remove old keys no longer held from old scan code array
   ldy #0
   nextRemove = *
   lda prevKeys,y        ;get current old key
   cmp #$ff
   beq ++
```

```
   ldx #rollover-1      ;search for it in the new scan code array
-  cmp newKeys,x
   beq +
   dex
   bpl -
   tya                  ;here, old key no longer held; remove it
   tax
-  lda prevKeys+1,x
   sta prevKeys+0,x
   inx
   cpx #rollover-1
   bcc -
   lda #$ff
   sta prevKeys+rollover-1
   sta ignoreKeys
+  iny                  ;check next old key
   cpy #rollover
   bcc nextRemove

   ;** insert new keys at front of old scan code array
+  ldy #0
   nextInsert = *
   lda newKeys,y        ;get current new key
   cmp #$ff
   beq ++
   ldx #rollover-1      ;check old scan code array for it
-  cmp prevKeys,x
   beq +
   dex
   bpl -
   pha                  ;it's not there, so insert new key at front, exit
   ldx #rollover-2
-  lda prevKeys+0,x
   sta prevKeys+1,x
   dex
   bpl -
   lda #0
   sta ignoreKeys
   pla
   sta prevKeys+0
   ldy #rollover        ;(trick to exit)
+  iny
   cpy #rollover
   bcc nextInsert
+  rts                  ;now, the head of the old scan code array contains
                        ;  the scan code to present to the Kernal, and other
                        ;  positions represent keys that are also held down
                        ;  that have already been processed and therefore can
                        ;  be ignored until they are released

checkJoystick = *       ;check if joystick is pushed: un-select all keyboard
   lda #$ff             ;  rows and see if there are any "0"s in the scan
   sta pa               ;  status register
   sta pk
-  lda pb
   cmp pb
   bne -
   cmp #$ff
   lda #$7f             ;restore to default Kernal row selected (to the one
   sta pa               ;  containing the STOP key)
   lda #$ff
   sta pk
   rts

;global variables

scanTable  .buf scanrows        ;values of the eleven keyboard scan rows
newKeys    .buf rollover        ;codes of up to three keys held simultaneously
ignoreKeys .buf 1               ;flag: if an old key has been released and no
                                ;  new key has been pressed, stop all key
                                ;  repeating
prevKeys   .buf rollover+2      ;keys held on previous scan
-----=-----
```

And that's all there is to it. :-)

5. THE C-64 KEYSCANNER

The boot program for the C-64 keyscanner is as follows:

```
10 d=peek(186)
```

```
20 if a=1 then 60
30 a=1
40 load"keyscan64",d,1
50 goto 10
60 sys 49152+5*256  : rem $c500
```

It is very much like boot programs for other machine language programs that don't load at the start of BASIC. It will load the binary from the last device accessed, and activate it.

A listing of the C-64 keyscanning code is not presented here because it is so similar to the C-128 listing. The only things that are different are the Kernal patches and the keyboard scanning (because the three extra rows don't have to be scanned). The IRQ had to be substantially copied from the ROM, again, to get at the call to the key scanning. Also, rather than taking over the BASIC reset vector (since there isn't one), the NMI vector is taken over to insure the survival of the key scanner after a RUN/STOP+RESTORE. A bit of its preamble also had to be copied out of ROM to get at the good stuff. If you want a copy of the C-64 listing, you can e-mail me.

6. UUENCODED FILES

Here are the binary executables in uuencoded form. The CRC32s of the four files are as follows:

```
crc32 = 3398956287 for "keyscan128"
crc32 = 2301926894 for "keyscan64.boot"
crc32 = 1767081474 for "keyscan64"
crc32 = 1604419896 for "keyshow"
```

```
begin 640 keyscan128
M`!5,'A;8(`H53&GZ.*T9T"D!\`>-&="EV,G_\&\L$=`P!"E`T#$XI=CP+"38
M4`:M-`J-$M"E`2G]"01(K2T*2*T1T"E_"2"HK1;0)-@P`RGO+`D0JM`HJ?^-
M$M"E`0D"*?L%V4BM+`I(K1'0*5^HK1;0*>^JL`BB!\K0_>KJJFB-&-!HA0&,
M$=".%M"P$ZTPT"D!\`REV"E`\`:M$=`0`3A8D`<@JA4@Y\8X8*D`C0#<C2_0
MK0'<S0'<T/C)__`Z((D7D#\@<18@B1>0-R"W%B#L%B`L%ZT^`X7,K3\#A<VB
M_RRT%S`QK;47R?_0!J73\":I6(74J&PZ`ZE_C0#<J?^-+]"I_Z("G;47RA#Z
M(-T6HO^I`(VT%ZE8A=2H3)?&(%46J4^@%HT`"HP!"J(`O3D6\`8@TO_HT/5@
M#4M%65-#04XQ,C@@24Y35$%,3$5$`"!5%DP#0'BI`Z`5C10#C!4#6*("J?^=
MM1?*$/JI`(VT%V"B_Z#_J?Z%S*G_A<U,F!:M`=S-`=S0^(P`W(POT$G_G:87
M.";,)LVES(T`W*7-C2_0Z.`+D-E@`08'!PJ`$"`$`0$!`@0((-T6H`2^J!:]
MIA<YK1;P$KFR%@73A=.YK19)_SVF%YVF%X@0X&"E`<4!T/I)_RE`2DJ%TV"B
M`JG_G;$7RA#ZH`"$S*(`AM2]IA?P'*342I`22(;-ILS@`[`&F)VQ%^;,ILUH
MR,D`T.88I=1I"(74Z.`+D--@H`"YM1?)__`DH@+=L1?P&,H0^)BJO;87G;47
MZ.`"D/6I_XVW%XVT%\C``Y#5H`"YL1?)__`FH@+=M1?P&LH0^$BB`;VU%YVV
M%\H0]ZD`C;07:(VU%Z`#R,`#D--@J?^-``-R-+]"M`=S-`=S0^,G_J7^-``-RI
9_XTOT&``````````````````````````````
`

end
begin 640 keyscan64.boot
M`0@."`H`1++"*#$X-BD`'0@4`(L@0;(Q(*<@-C``)0@>`$&R,0`Z""@`DR)+
M15E30T%.-C0B+$0L,0!#"#(`B2`Q,`!@"#P`GB`T.3$U,JHUK#(U-B`@.B"/
)("1#-3`P````````

end
begin 640 keyscan64
M`,5,Q,5,$,9,ZL4@ZO^ES-`IQLW0):D4A<VDTT;/KH<"L=&P$>;/A<X@).JQ
M\XV'`JZ&`J7.28`@'.JE`2D0\`J@`(3`I0$)(-`(I<#0!J4!*1^%`2!9Q4Q^
MZJD`C0#<K0'<S0'<T/C)__`Y(%C'D#D@6,8@6,>0,2"-QB"[QB#[QJF!A?6I
MZX7VHO\L>,<P+:UYQ\G_T`>MC0+P(:E`A<NH;(\"J7^-`-RI_Z("G7G'RA#Z
M(+7&HO^I`(UXQZE`A<NH3";K(.K%H@"]U<7P!B#2_^C0]6!+15E30T%.-C0@
M24Y35$%,3$5$#0!XJ0F@Q8T4`XP5`ZDGH,:-&`.,&0-8H@*I_YUYQ\H0^JD`
MC7C'8'BI,:#JC10#C!4#J4>@_HT8`XP9`UA@2(I(F$BI?XT-W:P-W3`?(`+]
MT`-L`H`@O/8@X?_0#R`5_2"C_2`8Y2#JQ6P"H$QR_J+_H/^I_H7U3';&K0'<
MS0'<T/B,`-Q)_YUMQS@F]:7UC0#<Z.`(D.-@`08'!X`0(`0!`0$($(+7&H`.^
M@<:];;<<YA<;P%+F)Q@V-`HV-`KF%QDG_/6W'G6W'B!#>>8*D`C8T"8*("J?^=
M=<?*$/J@`(3UH@"&R[UMQ!)(AO:F]3DL`\#8G77'YO6F]FC(R0#0YAB1RZ!0
MYABERVD(A<OHX`B0TV"@`+EUQ\G_\"+EYQ\G_\"2B`MUUQ_`8RA#XF*J]>L=
M]:G_C7O'C7C'R,`#D-6@`+EUQ\G_\":B`MUYQ_`:RA#X2*(!O7O'G7K'RA#W
MJ0"->,=HC7G'H`/(P`.0TV"I_XT`W*T!W,T!W-#XR?^I?XT`W&``````````
*````````````````
`

end
begin 640 keyshow
M`!-,"Q,``````````*F3(-+_(#83H@`@%Q0@5A.B"B`7%"!T$Z(4(!<4(/03
MHAX@%Q0@3!1,$!-XH@"I_HT`W*T!W,T!W-#X2?^=`Q,X+@#<Z.`(D.I88'BB
M!ZE_C0#<K0'<S0'<T/A)_YT#$SAN`-S*$.Q88'BI`(T"W*G_C0/<H`>I?X4#
MI0.-`=RM`-S-`-S0^*+_C@'<2?^B!PH^`Q/*$/DX9@.($-VI_XT"W*D`C0/<
M6&!XJ0"-`MRI_XT#W*``'J7^%`Z4#C0'<K0#<S0#<T/BB_XX!W$G_H@<*/@,3
MRA#Y.&8#B!#=J?^-`MRI`(T#W%@>*(`J?Z%`Z4#C0#<K0'<S0'<T/A)_YT#
M$S@F`^C@")#F6&"@!(8$(+T#$$R`V%!BE!&DHA020`N8%Z.`(D.I@A0*@@
```

~~~
6!T8"J0!I,)$$B!#U8````````*(`8```
`

end
~~~

===

In the Next Issue:

Next Issue:

Tech-tech - more resolution to vertical shift

One time half of the demos had pictures waving horizontally on the width of the whole screen. This effect is named tech-tech and it is done using character graphics. How exactly and is the same possible with sprites ?

THE DESIGN OF ACE-128/64

Design of ACE-128/64 command shell environment (and kernel replacement). This will cover the organization, internal operation, and the kernel interface of the still-under-development but possibly catching-on kernel replacement for the 128 and 64. The article will also discuss future directions and designs for the ACE environment. ACE has a number of definite design advantages over other kernel replacements, and a few disadvantages as well.