```
          __ __            ______
         |  ||__|  ______ /  __  \  ______  __ __  ______  ____  __ __
    _____|  | __  /  __  \|  |  |__|_/  __  \|  ||  | /  __  \ /    \|  ||  |
   /  __ /  ||  ||  |__|__||  |  |____/ |  |  ||  ||  ||  |__|  ||  ||__||  ||  |
  |  |  |   ||  |`\_____  \|  |  |_____\|  |  ||  ||  ||   _____||  | W |  ||  |
  |  |__|   ||  ||  |__|  ,||  |  |__|  ,||  |__|  ,||  ||  ,||  |__|  ,||  | a |  ||  |
  `\________||__|`\_______/'`\_______/'`\_______/'`\____/'`\_______/'| /' D `\    /'
         _     _  _ __ ___ ____ ______________________________|/_________|  |
          _     _  _ __ ___ ____ __________________________________________/'
```

The Journal of the Commodore Enthusiast

I s s u e 3 : March 26, 1997

P R E A M B L E

Welcome to the third issue of disC=overy, the Journal of the Commodore Enthusiast. We greet you proudly, the ones who still hold the beloved Commodore 8-bit machines in high regard and respect. We thank you from the bottom of our hearts and look forward to forging a solid productive relationship with the C= 8-bit community.

  - Mike Gordillo, Steven Judd, Ernest Stokes, George Taylor, and the
    authors of disC=overy.

```
::::::::::d:i:s:C=:o:v:e:r:y:::::::::::::::i:s:s:u:e::3::::::::::::::::::
::::::::::::::::::::::T A B L E   O F   C O N T E N T S::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

```
::::::::::d:i:s:C=:o:v:e:r:y:::::::::::::::i:s:s:u:e::3::::::::::::::::::
/S01::$A480::::::::::::::::::S O F T W A R E:::::::::::::::::::::::::::::::
```

::

```
Rommaging around : $A480-$A856
----------------              by
                                Stephen L. Judd  (sjudd@nwu.edu)
```

"Disassemble? ...Disassemble!!!" -- No. 5, 'Short Circuit'

        This series has a simple goal: to completely disassemble and document
the Commodore 64 ROMs. There is a nice ROM disassembly available on the
internet, with the actual hex code displayed and HTML hypertexted. This
version, on the other hand, is written in a more 'human readable' form,
heavily commented and labeled, and is intended to complement the other.
To that end, it does not duplicate very much information which is easily
obtainable elsewhere (ftp.funet.fi, or Marko Makela's homepage, for instance).

        BLARG is a program which adds several hires graphics commands to BASIC,
and was the main motivation to finally get started on this project. This
article will thus focus on the BASIC routines from $A480 (Main Loop) to
$A856 (END), with a goal of understanding how to add new commands to BASIC.

        The first part of the article gives an overview of BASIC and its
internal workings, and sets up some of the things used later on. The
second part discusses the parts of BASIC which relate specifically to
the disassembled ROM, with a focus on the vectored routines. The third
part gives a brief overview of BLARG. The ROM source, BLARG source,
and BLARG binaires are all included. These files are all available at

                http://stratus.esam.nwu.edu/~judd/fridge

        About the ROM listings: I did them up in Merlinesque format of
course. Probably the only unfamiliar thing is the concept of global
and local lables. Local labels are prefixed with a colon, and their
definition is only valid between two global lables. Otherwise they
may be redefined. Thus a piece of code like

```
PROC1   BNE :CONT
        INC TEMP1
:CONT   RTS

PROC2   BNE :CONT
        INY
:CONT   DEC TEMP1
        RTS
```

contains two global labels (PROC1 and PROC2). PROC1 will branch to the RTS
instruction and PROC2 will branch to the DEC TEMP1 instruction, since the
:CONT label is redefined once the global label PROC2 appears. The bottom line
is that if you see a branch to a local label, that label is nearby and
after the last global label.

BASIC overview

        Before starting, there are some important things to know about
BASIC. When you type in a line of text, and hit [RETURN], that text
is entered into a text buffer located at $0200. The text from this buffer
is then tokenized by the BASIC interpreter. If it is an immediate mode
command (doesn't start with a line number) that command is then executed,
otherwise it is stored in memory as a BASIC program line.
        BASIC lines are stored in memory as follows:

```
      ______________________________   _______________________  ...__
     /                              \ /                       \      \
0 [link] [line #] [Basic line] [0] [link] [line #] [Basic] [0]  ...  [0] [0]
|    |       |          |       |     |
|    |       |          |       |    Link to next line
|    |       |          |       |
|    |       |          |      End of line marker
|    |       |          |
|    |       |     Tokenized line of basic program
|    |       |
|    |     Two byte (lo,hi) line number
|    |
|  Two byte (lo,hi) pointer to next line in program
|
Beginning of program
```

        A zero marks the beginning of the program, which normally begins
at $0800 (2048). The next two bytes are the address in (lo,hi) form of

the next line in the program. The following two bytes are the line
number for the current line. Then the tokenized line of BASIC follows;
a null byte marks the end of the line. The next line follows immediately.
A line link of value 0 (actually only the high byte needs to be 0) marks
the end of the program.
        Thus a program like

```
        10 PRINT "HELLO"
```

will in memory look like

```
2048    0               ;Zero byte at beginning
2049    15              ;Next link = $080F = 15+8*256 = 2063
2050    8
2051    10              ;Line number = 10 + 0*256 = 10
2052    0
2053    153             ;PRINT token
2054    32              ;space
2055    34      "       ;quote
2056    72      H       ;PETSCII string
2057    69      E
2058    76      L
2059    76      L
2060    79      O
2061    34      "
2062    0               ;end of line
2063    0               ;end of program
2064    0
```

        Variables begin immediately following the end of the program.
Strings are stored beginning at the top of memory ($9FFF) and work
their way downwards towards the variables.

        Another very important feature of BASIC is that the important
routines are vectored. These are indeed filled vectors -- filled with
the address of the routines in question. The BASIC indirect vector
table is located at $0300-$030B:

```
IERROR  $0300-$0301  Vector to print BASIC error message routine
IMAIN   $0302-$0303  Vector to main BASIC program loop
ICRNCH  $0304-$0305  Vector to CRUNCH routine (tokenize ASCII text)
IQPLOP  $0306-$0307  Vector to QPLOP routine (tokens -> ASCII)
IGONE   $0308-$0309  Vector to GONE, executes BASIC tokens
IEVAL   $030A-$030B  Vector to routine which evaluates single-term math
                     expression
```

Calls to these routines are vectored through these addresses via an
indirect JMP -- they are called using JMP (IMAIN) for instance. These
vectors may then be redirected to new routines, and so they provide
a smooth way of adding new keywords to BASIC. CRUNCH is used to
tokenize lines of text when they are entered. QPLOP is used by
LIST to print tokens to ASCII text. GONE is used to execute tokens.
        Before modifying these vectors, keep in mind that many programs,
such as JiffyDOS, have already modified them!

        Finally, the routines CHRGET and CHRGOT are very important
in BASIC. They are copied into zero page when the system first starts
up, and are located at $0073:

```
$73     CHRGET  INC TXTPTR      ;Increment low byte
$75             BNE CHRGOT
$77             INC TXTPTR+1    ;High byte if necessary
$79     CHRGOT  LDA             ;Entry here doesn't increment TXTPTR
$7A     TXTPTR  $0207           ;low byte, high byte -- the LDA operand.
                                ;Generally points at BASIC or points
                                ;at input buffer at $0200 when in
                                ;immediate mode.
$7C     POINTB  CMP #$3A        ;Set carry if > ASCII 9
$7E             BCS EXIT        ;Exit if not a numeral
$80             CMP #$20        ;Check for ASCII space
$82             BEQ CHRGET      ;...skip space and move to next char
$84             SEC
$85             SBC #$30        ;Digits 0-9 are ASCII $30-$39
$87             SEC
$88             SBC #$D0        ;Carry set if < ASCII 0 ($30)
$8A     EXIT    RTS
```

On exit, the accumulator contains the character that was read; Carry clear
means character is ASCII digit 0-9, Carry set otherwise; Zero set if
character is a statement terminator 0 or an ASCII colon ($3A), otherwise
zero clear.

Wedge programs make this their entry point -- by redirecting CHRGET and CHRGOT a program can look at the input and act accordingly. Scanning text can get awfully slow, which is why most wedge programs you see use a single character to prefix wedge commands.

Finally, a list of BASIC tokens is a handy thing to have, along with the address of the routine which executes the statement:

$80	END	43057	$A831	
$81	FOR	42818	$A742	
$82	NEXT	44318	$AD1E	
$83	DATA	43256	$A8F8	
$84	INPUT#	43941	$ABA5	
$85	INPUT	43967	$ABBF	
$86	DIM	45185	$B081	
$87	READ	44038	$AC06	
$88	LET	43429	$A9A5	
$89	GOTO	43168	$A8A0	
$8A	RUN	43121	$A871	
$8B	IF	43304	$A928	
$8C	RESTORE	43037	$A81D	
$8D	GOSUB	43139	$A883	
$8E	RETURN	43218	$A8D2	
$8F	REM	43323	$A93B	
$90	STOP	43055	$A82F	
$91	ON	43339	$A94B	
$92	WAIT	47149	$B82D	
$93	LOAD	57704	$E168	
$94	SAVE	57686	$E156	
$95	VERIFY	57701	$E165	
$96	DEF	46002	$B3B3	
$97	POKE	47140	$B824	
$98	PRINT#	43648	$AA80	
$99	PRINT	43680	$AAA0	
$9A	CONT	43095	$A857	
$9B	LIST	42652	$A69C	
$9C	CLR	42590	$A65E	
$9D	CMD	43654	$AA86	
$9E	SYS	57642	$E12A	
$9F	OPEN	57790	$E1BE	
$A0	CLOSE	57799	$E1C7	
$A1	GET	43899	$AB7B	
$A2	NEW	42562	$A642	
$A3	TAB(			;Keywords which never begin a statement
$A4	TO			
$A5	FN			
$A6	SPC(			
$A7	THEN			
$A8	NOT			
$A9	STEP			
$AA	+	47210	$B86A	;Math operators
$AB	-	47187	$B853	
$AC	*	47659	$BA2B	
$AD	/	47890	$BB12	
$AE	^	49019	$BF7B	
$AF	AND	45033	$AFE9	
$B0	OR	45030	$AFE6	
$B1	>	49076	$BFB4	
$B2	=	44756	$AED4	
$B3	<	45078	$B016	
$B4	SGN	48185	$BC39	;Functions
$B5	INT	48332	$BCCC	
$B6	ABS	48216	$BC58	
$B7	USR	784	$0310	
$B8	FRE	45949	$B37D	
$B9	POS	45982	$B39E	
$BA	SQR	49009	$BF71	
$BB	RND	57495	$E097	
$BC	LOG	47594	$B9EA	
$BD	EXP	49133	$BFED	
$BE	COS	57956	$E264	
$BF	SIN	57963	$E26B	
$C0	TAN	58036	$E2B4	
$C1	ATN	58126	$E30E	
$C2	PEEK	47117	$B80D	
$C3	LEN	46972	$B77C	
$C4	STR$	46181	$B465	
$C5	VAL	47021	$B7AD	
$C6	ASC	46987	$B78B	

```
$C7     CHR$    46828 $B6EC
$C8     LEFT$   46848 $B700
$C9     RIGHT$  46892 $B72C
$CA     MID$    46903 $B737

$CB     GO                     ;Makes GO TO legal.
```

BASIC ROM

        The BASIC ROM begins at $A000:

```
$A000-$A001     Cold start vector
$A002-$A003     Warm start vector
$A004-$A00B     ASCII text "CBMBASIC"

$A00C-$A051     Statement dispatch table
```

        This is the first of several important tables used by BASIC.
When a new BASIC statement is to be executed the interpreter looks
here to find the address of the statement routine. Each entry is
a two-byte vector containting the routine address minus one. The
address is pushed onto the stack, so that the next RTS will jump
to the correct location. The addresses are in token order, beginning
with token $80 (END) and ending with token $A2 (NEW):

```
$A052-$A07F     Function dispatch vector table
```

        This is another table of two-byte vectors for BASIC functions --
that is, commands which are followed by an argument inside of parenthesis,
for example INT(3.14159). The entries are again in token order beginning
with token $B4 (SGN) and ending with token $CA (MID$).

```
$A080-$A09D     Operator dispatch vector table
```

        This table is for math operators, beginning with token $AA (+)
and ending with token $B3 (<).

```
$A09E-$A19D     List of keywords
```

        This is a table of all the reserved BASIC keywords. The
high bit of the last character set, so it is easy to detect the end
of a keyword. The keywords are listed in token order, so to find
the token corresponding to a given keyword the BASIC interpreter
simply moves down this list, counting as it goes. If a match
occurs, then the token value is simply the counter value.
        The table is searched using SBC instead of CMP. If the
result of the subtraction is $80 -- keywords end with the high bit set --
then a valid keyword has been found. When input is placed into the
input buffer at $0200, character codes 192-223 are used for shifted
characters. From this it should be clear why the keyboard shortcuts work,
for instance typing pO instead of poke. It should also be clear why poK will
also work, but pokE will not work.

```
$A19E-$A327     ASCII text of BASIC error messages (dextral character inverted)
$A328-$A364     Error message vector table
$A365-$A389     Miscellaneous messages (null-terminated)

$A38A-$A47F     Some BASIC routines, to be be disassembled at a later date

$A480           Main loop
```

        Since the goal is to get a good enough understanding of BASIC to
add new keywords, the main program loop is a good place to start. This
routine is vectored through IMAIN at $0302. It gets a line of input
from the keyboard, checks for a line number, and processes the line
appropriately.
        When I started programming BLARG, I had no idea what routines like
CRUNCH were expected to return -- is the Y register expected to have
a certain value upon exit? Or maybe a certain variable needs to be
set up so that another routine may reference it? Thus it is a good
idea to begin at the main loop and see how a line is normally
processed. The disassembly listing begins at this point.

        Other important vectored routines are:

```
$A579           CRUNCH
```

        This routine goes through the BASIC text buffer at $0200 and
tokenizes any keywords which aren't in quotes. As the routine begins
to scan the input buffer it discards any characters which have their

high bit set, such as shifted characters. (This is only in the
initial search for keywords -- shifted characters within quotes or
as part of a keyword are taken care of by another routine). As
a practical matter, this means that any routine which adds extra
tokens must call this routine _first_, since it will just skip over
custom tokens otherwise!
        When it is finished processing the input buffer, the
input buffer contains the tokenized line, terminated by a null
byte, _and_ with another null byte at the end of the line +2 (much
like the terminating byte which marks the end of a program). See
the disassembly and blarg source for other things which are set up.

$A717   QPLOP

        This is the part of the LIST routine (which begins at $A69C)
which converts tokens into ASCII characters and prints them to the
screen.

$A7E4   GONE

        This is the routine which gets the next token (every statement
begins with a token or an implied LET) and executes the appropriate command.
Invalid tokens generate a SYNTAX ERROR. Character values less than 128 cause
LET to be executed.
        The IF/THEN routine actually bypasses the IGONE vector and jumps
directly into this routine, which means that lines like
        IF A=0 THEN MYCOMMAND
will generate a syntax error. BLARG currently has no mechanism for
getting around this (because I did not discover this until recently,
and the article deadline was several days ago :), except to place a
colon after the THEN, i.e. IF A=0 THEN:MYCOMMAND. Apparently some
programs get around this by defining their own IF statement; probably
the best way to get around it is to redirect the IERROR vector.
When an error is generated, check to see if it is a custom token,
then check to see if it was preceded by a THEN token.

BLARG

        By now, we have all the necessary information to add new keywords
to BASIC. To that end I present BLARG, which adds several hires bitmap
commands to BASIC. As a bonus, BLARG can take advantage of a SuperCPU
if you have one. These commands are much faster than the 128's BASIC7.0
commands and in my quite biased opinion much more intuitive. The CIRCLE
and LINE commands are adaptations of my routines from C=Hacking; look there
for more info on the actual routines.
        Some things in BLARG are done a little differently than their
corresponding BASIC routines, in large part because I wrote some of the
routines before disassembling the BASIC ones.
        BLARG documentation is below.

References

        I have found Mapping the 64 to be an invaluable reference, along
with "Programming the Commodore 64", by Raeto West. ftp.funet.fi has
some good documentation, and I use Marko Makela's web page quite
often at http://www.hut.fi/~msmakela -- there are many useful documents
there, including a complete html cross-referenced ROM listing.

```
*
* A480-A856  (MAIN-END)
*   Basic ROM disassembly starting with main loop
*
* Stephen L. Judd 1997
*

*
* Labels are at the end of the listing, along with a list
* of major routines and their addresses.
*

*
* MAIN -- Main loop, receives input and executes
* immediately or stores as a program line.
*

* $A480
JMPMAIN  JMP (IMAIN)      ;Main loop, below
                          ;Vectored through IMAIN, so it
                          ;may be redirected.
```

```
MAIN     JSR INLIN         ;Input a line from keyboard
         STX TXTPTR
         STY TXTPTR+1
         JSR CHRGET        ;Get 1st char out of buffer
         TAX
         BEQ JMPMAIN       ;Empty line
         LDX #$FF          ;Signal that BASIC is in immediate
         STX CURLIN+1      ;mode.
         BCC MAIN1         ;CHRGET clears C when digit is read
         JSR CRUNCH        ;Tokenize line
         JMP JMPGONE       ;Execute line

*
* MAIN1 -- Add or replace a line of program text.
*
* TEMP2 points to the current line to be deleted
* TEMP1 will point to the start of memory to be copied
* INDEX1 will point to the destination for the copy

* $A49C
MAIN1    JSR LINGET        ;Convert ASCII to binary line number
         JSR CRUNCH        ;Tokenize buffer
         STY COUNT         ;Y=length of line
         JSR FINDLINE      ;Find the address of the line number
         BCC :NEWLINE
         LDY #$01          ;Replace line of text
         LDA (TEMP2),Y     ;$5F = pointer to line
         STA TEMP1+1       ;(TEMP1) = xx, Hi byte of next line
         LDA VARTAB        ;End of BASIC program
         STA TEMP1         ;(TEMP1) = basic end, high next
         LDA TEMP2+1
         STA INDEX1+1      ;(INDEX1) = xx, Hi byte current line
         LDA TEMP2         ;Compute -length of current line
         DEY
         SBC (TEMP2),Y     ;Current address - next address
         CLC
         ADC VARTAB        ;end - (length of line to be deleted)
         STA VARTAB
         STA INDEX1        ;(INDEX1) = bytes to delete, high current
         LDA VARTAB+1      ;Back up end of program if necessary
         ADC #$FF
         STA VARTAB+1
         SBC TEMP2+1       ;Number of pages of memory to move
         TAX
         SEC
         LDA TEMP2         ;Pretty confusing, eh?
         SBC VARTAB
         TAY               ;256 - number of bytes to move
         BCS :CONT1
         INX
         DEC INDEX1+1
:CONT1   CLC
         ADC TEMP1         ;old basic end - number of bytes to move
         BCC :CONT2
         DEC TEMP1+1
         CLC
:CONT2   LDA (TEMP1),Y     ;Delete old line, fall through to create
         STA (INDEX1),Y    ;new line.
         INY
         BNE :CONT2
         INC TEMP1+1
         INC INDEX1+1
         DEX
         BNE :CONT2

:NEWLINE JSR RESCLR        ;Reset variables and TXTPTR
         JSR LINKPRG       ;Relink program lines
         LDA BUF
         BEQ JMPMAIN       ;Empty statement -- nothing to do
         CLC
         LDA VARTAB        ;Start of variables
         STA $5A
         ADC COUNT
         STA $58           ;Start of vars after line is added
         LDY VARTAB+1
         STY $5B
         BCC :CONT3
         INY
:CONT3   STY $59
         JSR MALLOC        ;Open up some space in memory
         LDA LINNUM        ;Number of line to be added
         LDY LINNUM+1
         STA H01FE          ;Rock bottom on the stack.
         STY H01FF
         LDA STREND         ;Bottom of strings to top of variables
         LDY STREND+1
         STA VARTAB
         STY VARTAB+1
         LDY COUNT          ;Number of chars in BUF
         DEY
:LOOP    LDA BUF-4,Y        ;Copy buffer contents into program
         STA (TEMP2),Y
         DEY
         BPL :LOOP
         JSR RESCLR
         JSR LINKPRG
         JMP JMPMAIN        ;Wheeeeeeee...

*
* LINKPRG -- Relink lines of program text
*
* $A533
LINKPRG  LDA TXTTAB         ;Start of BASIC text
         LDY TXTTAB+1
         STA TEMP1
         STY TEMP1+1
         CLC
:LOOP1   LDY #$01
         LDA (TEMP1),Y
         BEQ :RTS           ;0 means end of program
         LDY #$04           ;Skip link, line number, and 1st char
:LOOP2   INY
         LDA (TEMP1),Y
         BNE :LOOP2         ;Find end of line
         INY
         TYA
         ADC TEMP1          ;Add offset to pointer to get address
         TAX                ;of next line
         LDY #$00
         STA (TEMP1),Y      ;And store in link
         LDA TEMP1+1
         ADC #$00
         INY
         STA (TEMP1),Y
         STX TEMP1          ;Move to next line
         STA TEMP1+1
         BCC :LOOP1         ;Keep on truckin'!
:RTS     RTS

*
* INLIN -- Input a line from keyboard to buffer
*
* $A560
INLIN    LDX #$00
:LOOP    JSR HE112          ;BASIC's way of calling Kernal routines
         CMP #$0D
         BEQ :DONE
         STA BUF,X
         INX
         CPX #$59           ;Maximum buffer size = 88 chars
         BCC :LOOP
         LDX #$17           ;STRING TOO LONG error
         JMP ERROR
:DONE    JMP HAACA          ;Part of the PRINT routine

*
* CRUNCH -- Tokenize a line of text contained in the input buffer
*
* $A579
*
* On exit:
*   Y = last character of crunched text + 4
*   X = last char of input buffer read in.
*   TEXTPTR = $01FF
*   GARBFL = 00 if colon was read, $49 if DATA statement,
*            04 otherwise.
*   ENDCHR = $22 (if quote was found) or 0 (if REM)
*   COUNT = Last token read - 128 (contrary to what
*                                  Mapping the 64 says)
*   TEMP1 = More or less random
*   BUF = Tokenized text, followed by a 00, random byte, and 00
*
```

```
CRUNCH                          ;$A579
         JMP (ICRNCH)
         LDX TXTPTR             ;Low byte
         LDY #$04
         STY GARBFL

MAINLOOP LDA BUF,X              ;Search input buffer for text
         BPL :GOTCHAR           ;characters or #$FF
         CMP #$FF
         BEQ STALOOP
         INX
         BNE MAINLOOP

:GOTCHAR CMP #$20               ;Is it a space?
         BEQ STALOOP
         STA ENDCHR
         CMP #$22               ;Is it a quote?
         BEQ QUOTE2
         BIT GARBFL             ;This either equals 00 if a
                                ;colon was hit, $49 if a DATA
                                ;statement was found, and #04
                                ;initially.  Thus, it prints the
                                ;text within DATA statements.
         BVS STALOOP
         CMP #'?'               ;Short print
         BNE :CONT1
         LDA #$99               ;Print token
         BNE STALOOP

:CONT1   CMP #'0'               ;Check for a number,
         BCC :NOTNUM
         CMP #'<'               ;colon, or semi-colon
         BCC STALOOP

:NOTNUM  STY TEMP1              ;So, search for a keyword
         LDY #$00
         STY COUNT
         DEY
         STX TXTPTR
         DEX

FINDWORD INY                    ;Find keyword
         INX
CMPWORD  LDA BUF,X              ;Match against keyword table
         SEC
         SBC RESLST,Y
         BEQ FINDWORD
         CMP #$80               ;High bit of last char is set
                                ;Also means p shift-O gives short
                                ;version of POKE (for instance),
                                ;as should po shift-K.
         BNE NEXTWORD
         ORA COUNT              ;A now contains token value
STABUF   LDY TEMP1              ;Crunched text index
STALOOP  INX                    ;Store text in crunched buffer
         INY
         STA BUF-5,Y
         LDA BUF-5,Y            ;Goofy -- use AND #$FF or CMP #0
         BEQ ALLDONE            ;Zero byte terminates string
         SEC
         SBC #$3A               ;Is it a colon?
         BEQ :COLON
         CMP #$49               ;Was it $83, a DATA statement?
         BNE :SKIP
:COLON   STA GARBFL
:SKIP    SEC
         SBC #$55               ;Was it $8F, a REM statement?
         BNE MAINLOOP
         STA ENDCHR             ;If REM, then read rest of line

QUOTE    LDA BUF,X              ;Look for matching quote char
         BEQ STALOOP            ;or end of statement, and embed
         CMP ENDCHR             ;text directly.
         BEQ STALOOP
QUOTE2   INY
         STA BUF-5,Y
         INX
         BNE QUOTE

NEXTWORD LDX TXTPTR             ;Skip to next keyword
```

```
         INC COUNT        ;Next token
:LOOP    INY
         LDA RESLST-1,Y   ;Find last char
         BPL :LOOP
         LDA RESLST,Y
         BNE CMPWORD
         LDA BUF,X
         BPL STABUF

ALLDONE  STA BUF-3,Y
         DEC TXTPTR+1
         LDA #$FF
         STA TXTPTR
         RTS

*
* FINDLINE
*   Search for the line number contained in $14.  If found, set $5F
*   to link address and set carry.  Carry clear means line number
*   not found.
*

* $A613
FINDLINE LDA TXTTAB       ;Start of program text
         LDX TXTTAB+1
:LOOP    LDY #$01
         STA TEMP2
         STX TEMP2+1
         LDA (TEMP2),Y
         BEQ :EXIT        ;Exit if at end of program
         INY
         INY
         LDA LINNUM+1     ;High byte
         CMP (TEMP2),Y
         BCC :RTS         ;Less than -> line doesn't exist
         BEQ :CONT1
         DEY
         BNE :CONT2       ;...always taken
:CONT1   LDA LINNUM       ;Compare low byte
         DEY
         CMP (TEMP2),Y
         BCC :RTS         ;Punt when past line number
         BEQ :RTS         ;Success!
:CONT2   DEY              ;Get next line link
         LDA (TEMP2),Y
         TAX
         DEY
         LDA (TEMP2),Y
         BCS :LOOP
:EXIT    CLC
:RTS     RTS

* Perform NEW
* $A642
NEW      BNE *-2          ;RTS above
         LDA #$00
         TAY
         STA (TXTTAB),Y   ;Zero out first two bytes of program
         INY
         STA (TXTTAB),Y
         LDA TXTTAB
         CLC
         ADC #$02
         STA VARTAB       ;Move end of BASIC to begin+2
         LDA TXTTAB+1
         ADC #$00
         STA VARTAB+1
RESCLR   JSR RUNC         ;Reset TXTPTR
         LDA #$00

* Perform CLR (Calcium Lime and Rust remover!)
* $A65E
CLEAR    BNE CLEAREND
         JSR CLALL        ;Close files
         LDA MEMSIZ       ;Highest address used by BASIC
         LDY MEMSIZ+1
         STA FRETOP       ;Bottom of string text storage
         STY FRETOP+1
         LDA VARTAB       ;End of BASIC program/variable start
         LDY VARTAB+1
         STA ARYTAB       ;Start of array storage
         STY ARYTAB+1
         STA STREND       ;End of array storage/start of free RAM
         STY STREND+1
         JSR RESTORE
         LDX #$19
         STX TEMPPT       ;Temporary string stack
         PLA
         TAY
         PLA
         LDX #$FA         ;Reset stack
         TXS
         PHA              ;Restore correct return address
         TYA
         PHA
         LDA #$00
         STA OLDTXT+1     ;Address of current basic statement
         STA SUBFLG       ;
CLEAREND RTS

*
* RUNC -- reset current text character pointer to the
* beginning of program text.
*
* $A68E
RUNC     CLC
         LDA TXTTAB
         ADC #$FF
         STA TXTPTR
         LDA TXTTAB+1
         ADC #$FF
         STA TXTPTR+1
         RTS

*
* LIST -- perform LIST
* entered via return from CHRGET
*
* $A69C
LIST     BCC :SKIP        ;Is next char an ASCII digit
         BEQ :SKIP        ;Is next char ':' or 00
         CMP #$AB         ;Is next char '-' (token)
         BNE CLEAREND     ;RTS, above
:SKIP    JSR LINGET       ;Read decimal, convert to number
         JSR FINDLINE     ;Find line number
         JSR CHRGOT       ;Get char again
         BEQ :CONT        ;statement terminator
         CMP #$AB
         BNE NEW-1        ;RTS
         JSR CHRGET       ;Advance and get end of
         JSR LINGET       ;range to list xx-xx
         BNE NEW-1        ;RTS
:CONT    PLA
         PLA
         LDA LINNUM
         ORA LINNUM+1
         BNE LISTLOOP
         LDA #$FF         ;Signal to list program to end
         STA LINNUM
         STA LINNUM+1
LISTLOOP LDY #$01
         STY GARBFL
         LDA (TEMP2),Y
         BEQ ENDLIST
         JSR TESTSTOP     ;Test for STOP key
         JSR HAAD7        ;part of PRINT routine
         INY
         LDA (TEMP2),Y
         TAX
         INY
         LDA (TEMP2),Y
         CMP LINNUM+1     ;Check to see if at last line
         BNE :CONT1
         CPX LINNUM
         BEQ :CONT2
:CONT1   BCS ENDLIST
:CONT2   STY FORPNT       ;temporary storage
         JSR LINPRT       ;Print line number
         LDA #$20         ;space
LISTENT1 LDY FORPNT
         AND #$7F         ;strip high bit
LISTENT2 JSR HAB47        ;Prints char, AND #$FF
```

```
         CMP #$22          ;Look for a quote
         BNE :CONT
         LDA GARBFL
         EOR #$FF
         STA GARBFL
:CONT    INY               ;$A700
         BEQ ENDLIST       ;256 character lines perhaps?
         LDA (TEMP2),Y
         BNE JMPPLOP       ;Print the token
         TAY               ;Get line link
         LDA (TEMP2),Y
         TAX
         INY
         LDA (TEMP2),Y
         STX TEMP2
         STA TEMP2+1
         BNE LISTLOOP
ENDLIST  JMP HE386         ;Exit through warm start

*
* QPLOP -- print BASIC tokens as ASCII characters.
*
* $A717
JMPPLOP  JMP (IQPLOP)
QPLOP    BPL LISTENT2      ;Exit if not a token
         CMP #$FF
         BEQ LISTENT2
         BIT GARBFL
         BMI LISTENT2      ;Exit if inside a quote
         SEC
         SBC #$7F          ;Table offset+1
         TAX
         STY FORPNT        ;Temp storage
         LDY #$FF
:LOOP1   DEX               ;Traverse the keyword table
         BEQ :PLOOP
:LOOP2   INY               ;read a keyword
         LDA RESLST,Y
         BPL :LOOP2
         BMI :LOOP1
:PLOOP   INY               ;Print out the keyword
         LDA RESLST,Y
         BMI LISTENT1      ;Exit if on last char
         JSR HAB47         ;Print char, AND #$FF
         BNE :PLOOP

*
* Perform FOR
*
* $A742
FOR      LDA #$80
         STA SUBFLG
         JSR LET
         JSR FINDFOR
         BNE :CONT
         TXA
         ADC #$0F
         TAX
         TXS
:CONT    PLA
         PLA
         LDA #$09
         JSR CHKSTACK
         JSR ENDSTAT
         CLC
         TYA
         ADC TXTPTR
         PHA
         LDA TXTPTR+1
         ADC #$00
         PHA
         LDA CURLIN+1
         PHA
         LDA CURLIN
         PHA
         LDA #$A4
         JSR CHKCOM
         JSR HAD8D
         JSR FRMNUM
         LDA FACSGN
         ORA #$7F
         AND $62
         STA $62
         LDA #$8B
         LDY #$A7
         STA TEMP1
         STY TEMP1+1
         JMP HAE43

         LDA #$BC
         LDY #$B9
         JSR MTOFAC
         JSR CHRGOT
         CMP #$A9
         BNE HA79F
         JSR CHRGET
         JSR FRMNUM
HA79F    JSR SIGN
         JSR HAE38
         LDA $4A
         PHA
         LDA FORPNT
         PHA
         LDA #$81
         PHA

*
* NEWSTT -- Set up next statement for execution.
*
* $A7AE
NEWSTT   JSR TESTSTOP
         LDA TXTPTR
         LDY TXTPTR+1
         CPY #$02          ;Text buffer?
         NOP
         BEQ :CONT
         STA OLDTXT
         STY OLDTXT+1
:CONT    LDY #$00          ;End of last statement.
         LDA (TXTPTR),Y
         BNE HA807         ;branch into GONE
         LDY #$02
         LDA (TXTPTR),Y    ;End of program?
         CLC
         BNE :CONT2
         JMP HA84B         ;exit through END

:CONT2   INY
         LDA (TXTPTR),Y    ;Line number
         STA CURLIN
         INY
         LDA (TXTPTR),Y
         STA CURLIN+1
         TYA
         ADC TXTPTR        ;Advance pointer
         STA TXTPTR
         BCC JMPGONE
         INC TXTPTR+1

*
* GONE -- Read and execute next statment
*
* $A7E1
JMPGONE  JMP (IGONE)
         JSR CHRGET
         JSR GONE
         JMP NEWSTT
* $A7ED
GONE     BEQ :RTS          ;Exit if statement terminator
         SBC #$80
         BCC :JMPLET       ;If not a token then a variable
         CMP #$23          ;Tokens above $A3 never begin
         BCS CHKGOTO       ;a statement (except GO TO)
         ASL               ;Otherwise, get entry point
         TAY               ;of the statement...
         LDA STATVEC+1,Y
         PHA
         LDA STATVEC,Y
         PHA
         JMP CHRGET        ;... so CHRGET can RTS to it.

:JMPLET  JMP LET
```

```
HA807    CMP #':'
         BEQ JMPGONE
JMPSYN   JMP SYNERR

CHKGOTO  CMP #$4B           ;Is it "GO TO"?
         BNE JMPSYN
         JSR CHRGET
         LDA #$A4           ;Make sure next char is "TO"
         JSR CHKCOM         ;token, and skip it.
         JMP GOTO

*
* Perform RESTORE
*
* $A81D
RESTORE  SEC
         LDA TXTTAB         ;Set DATA pointer to start of program
         SBC #$01
         LDY TXTTAB+1
         BCS :CONT
         DEY
:CONT    STA DATPTR         ;Address of current DATA item
         STY DATPTR+1
:RTS     RTS

TESTSTOP JSR STOP
         BCS END+1

*
* END -- perform END
*
* $A831
END      CLC
         BNE $A870          ;RTS
         LDA TXTPTR
         LDY TXTPTR+1
         LDX CURLIN+1       ;Current line number
         INX
         BEQ :CONT1         ;Branch if in immediate mode
         STA OLDTXT         ;Save statement address for CONT
         STY OLDTXT+1
         LDA CURLIN
         LDY CURLIN+1
         STA OLDLIN         ;Restored by CONT
         STY OLDLIN+1
:CONT1   PLA                ;Discard return address
         PLA
* $A84B
HA84B    LDA #$81           ;Message at $A381
         LDY #$A3
         BCC :CONT2
         JMP $A469          ;BREAK
:CONT2   JMP $E386          ;BASIC warm start



ENDCHR   = $0008            ;See CRUNCH
COUNT    = $0B
GARBFL   = $000F            ;Work byte
SUBFLG   = $0010
LINNUM   = $14
TEMPPT   = $0016            ;Next available space in temp string stack
TEMP1    = $22              ;Temporary pointer/variable
INDEX1   = $0024
TXTTAB   = $002B            ;Pointer to start of BASIC text
ARYTAB   = $002F            ;Pointer to start of arrays
STREND   = $0031            ;End array storage/start free RAM
FRETOP   = $0033            ;End of string text/top free RAM
MEMSIZ   = $0037            ;Highest address used by BASIC
CURLIN   = $39              ;Current BASIC line number
OLDTXT   = $003D            ;Pointer to current BASIC statement
DATPTR   = $0041            ;Pointer to current DATA item
FORPNT   = $0049            ;Temp pointer to FOR index variable
TEMP2    = $5F
FAC1     = $61
FACSGN   = $0066
TEMP1    = $71
FBUFPT   = $0071
CHRGET   = $0073            ;Get next BASIC text char
CHRGOT   = $0079            ;Get current BASIC text char
```

```
TXTPTR   = $7A             ;Text pointer
H01FE    = $01FE
H01FF    = $01FF
BUF      = $0200           ;Text input buffer
IMAIN    = $0302           ;System vectors
ICRNCH   = $0304
IQPLOP   = $0306
IGONE    = $0308
STATVEC  = $A00C           ;Statement dispatch vector table
RESLST   = $A09E           ;Reserved keywords list
FINDFOR  = $A38A           ;Find FOR on stack
MALLOC   = $A3B8           ;Make space for new line or var
CHKSTACK = $A3FB           ;Check for space on stack
ERROR    = $A437           ;General error handler
END      = $A831           ;Perform END
GOTO     = $A8A0           ;Perform GOTO
ENDSTAT  = $A906           ;Search for end of statement
                           ;(00 or colon)
LINGET   = $A96B           ;Convert ASCII decimal to 2-byte line number
LET      = $A9A5           ;Perform LET
HAACA    = $AACA
HAAD7    = $AAD7
HAB47    = $AB47
FRMNUM   = $AD8A           ;Eval expression/check data type
HAD8D    = $AD8D
HAE38    = $AE38
HAE43    = $AE43
CHKCOM   = $AEFF           ;Check for and skip comma
SYNERR   = $AF08           ;Print Syntax Error message
MTOFAC   = $BBA2           ;Move FP number from mem to FAC1
SIGN     = $BC2B           ;Sign of FAC1 in A
LINPRT   = $BDCD           ;Print num X,A=lo,hi in ASCII
HE112    = $E112
HE386    = $E386
STOP     = $FFE1           ;Kernal
CLALL    = $FFE7

*
* Major routine entry points
*
* $A480 JMPMAIN  JMP (IMAIN)       ;Main loop, below
* $A483 MAIN     JSR INLIN         ;Input a line from keyboard
* $A49C MAIN1    JSR LINGET        ;Convert ASCII to binary line number
* $A533 LINKPRG  LDA TXTTAB        ;Start of BASIC text
* $A560 INLIN    LDX #$00
* $A579 CRUNCH   JMP (ICRNCH)
* $A613 FINDLINE LDA TXTTAB        ;Start of program text
* $A642 NEW      BNE *-2           ;RTS above
* $A65E CLEAR    BNE CLEAREND
* $A68E RUNC     CLC
* $A69C LIST     BCC :SKIP         ;Is next char an ASCII digit
* $A717 JMPPLOP  JMP (IQPLOP)
* $A71A QPLOP    BPL LISTENT2      ;Exit if not a token
* $A7AE NEWSTT   JSR TESTSTOP
* $A7E1 JMPGONE  JMP (IGONE)
* $A7ED GONE     BEQ :RTS          ;Exit if statement terminator
* $A81D RESTORE  SEC
* $A831 END      CLC
```

BLARG -- Basic Language Graphics extension
-----
version 1.0 2/10/97

BLARG is a little BASIC extension which adds some graphics commands to the normal C-64 BASIC. In addition it supports the SuperCPU optimization modes, as well as double buffering. Finally, it is free for use in your own programs, so feel free to do so!

My goal was to write a BASIC extension which was compact, fast, and actually available for downloading :). Also, something that wasn't BASIC7.0. It doesn't have tons of features but I deem it to be Nifty.

Anyways, these are adaptations of my algorithms from C=Hacking and such. They are not the most efficient implementations, but they are fairly zippy, and fairly well beat the snot out of BASIC7.0 commands! For instance, the times from moire3 (a line drawing test):

```
        Stock 64        1200 jiffies    (1X)
        SCPU Mode 17    137 jiffies     (9.1X)
        SCPU Mode 16    59 jiffies      (20.2X)
        BASIC7.0 (1MHz) 4559 jiffies    (1/3.5 X)
```

So lines are nearly 4x faster on a stock 64 than a 128 running BASIC7.0.
Running in mode16 on a SuperCPU is 77 times faster than BASIC7.0!!!

Let's not even talk about BASIC7.0 circles. Well, what the heck, let's
talk about them :)
circletest1:

```
        Stock 64        360 jiffies     (1x)
        SCPU Mode 17    50 jiffies      (7x)
        SCPU Mode 16    22 jiffies      (15.4x)
        BASIC7.0        17394 j :)      1/49x
```

Bottom line: mode 16 circles are, if you can believe it, 790x faster
than BASIC7.0 circles (and much better looking, especially at large
radii -- BASIC7.0 circles are actually 128-sided polygons :-/).

The total size of the program right now is a bit over 2k, and sits at
$C000. To install the program, just load and run. To re-initialize
the system (after a warm reset for instance) just type SYS49152. The
command list is located near $C000, immediately followed by the
routine addresses, in case you want to take a peek.

A second program, BLARG$8000, is included in the archive. To use it,
load ,8,1 and type SYS32768 to initialize. This program is included
so that it may be loaded from a BASIC program.

Several demo programs are included, and offer a good way of learning
the commands (for instance, try typing ORIGIN 10,10 before running
MOIRE3).

Words to the wise:

1 - If you use a DOS extension like JiffyDOS then be sure to
    enclose filenames etc. in quotes (unless you want them to
    be tokenized, in which case feel free to omit quotes).

2 - If it looks like your machine has completely frozen try
    typing RUN-STOP, shift-clear, MODE 17 (Sometimes you can
    break the program before it can tell VIC where the screen
    is located). MODE16 and MODE17 will always fix stuff up,
    whereas run/stop-restore doesn't always do the trick.

3 - Always keep in mind that MODE 16,17, and 18 may hose
    string variables.

4 - IF/THEN bypasses the IGONE vector, so a statement like
    IF A=0 THEN GRON will fail. The statement IF A=0 THEN:GRON
    will work fine.

Without further ado:

GRON [COLOR] -- Turns graphics on. If the optional parameter COLOR is
specified the bitmap is cleared and the colormap is initialized to
that value, specifically,
COLOR = 16*foreground color + background color

```
        Examples:
                GRON  -- Turn on bitmap without clearing it.
                GRON20 -- Turn on bitmap, clear, purple bg, white fg
```

GROFF -- Turns graphics off (sticks you back into text mode, or
whatever mode you were in when you last called GRON)

CLEAR [color] -- Clear current graphics buffer. CLEAR is part of
the GRON routine, but will not set VIC or CIA#2.

COLOR n -- COLOR 1 sets the drawing color to the foreground color;
COLOR 0 sets it to background.

ORIGIN CX, CY -- Sets the upper-left corner of the screen to have
coordinates CX,CY. More precisely, commands will subtract
CX,CY from coordinates passed into it. Among other things,
this provides a mechanism for negative numbers to be
handled -- LINE -10,0,40,99 will not work, but ORIGIN 10,0:
LINE 0,0,50,99 will. This value is initialized to (0,0) whenever
SYS49152 is called.

PLOT X,Y -- Sticks a point at coordinates X,Y. (Actually at
coordinates X-XC, Y-YC). X may be in the range 0..319
and Y may be 0..199; points outside this range will not be

plotted.

LINE X1,Y1,X2,Y2 -- Draws a line from X1,Y1 to X2,Y2 (subtracting XC,YC as necessary). X1 may be any 16-bit value and Y2 may be any 8-bit value. Coordinates off of the screen will simply not be plotted!

CIRCLE XC,YC,R -- Draws a circle of radius R centered at XC,YC (translating as necessary). The algorithm is smart and can handle R=0..255 correctly (as far as I know!). I used a modified version of my algorithm, which makes very nice circles in my quite biased opinion, except for a few radii which come out a little ovalish. Oh well.

```
MODE n -- New graphics MODE.
        MODE 16 -- SuperCPU mode.  This moves the bitmap screen to
                $A000, the colormap to $8C00, the text screen to $8800,
                and sets the SuperCPU bank 2 optimization mode.
                $9000-$9FFF is totally unused :(.
        MODE 17 -- Normal mode.  Bitmap->$E000, Colormap ->$CC00,
                text screen -> $0400.
        MODE 18 -- Double buffer mode.  MODE16 memory configuration,
                no SCPU optimization.
        MODE16, MODE17, and MODE18 reset the BASIC memtop and stringtop
        pointers, so any defined strings may get hosed.
        (They also execute GROFF, and hence turn on the text screen).
        Any other MODE parameter will be set to the BITMASK parameter.
        What is BITMASK?  Anything drawn to the screen is first ANDed
        with BITMASK.  (Try MODE 85 sometime).
        (Although these numbers are reserved for future expansion).
```

BUFFER n -- Set drawing buffer.
When double-buffer mode is activated (MODE 18), both buffers are available for drawing and displaying. It is then possible to draw in one buffer while displaying the other. BUFFER n selects which buffer the PLOT,LINE,CIRCLE, and CLEAR commands will affect. If n=0 then it swaps the target buffer. Otherwise, n=odd references the buffer at $A000 and n=even selects the buffer at $E000.

SWAP -- Swap displayed buffer.
Assuming MODE18 is selected, SWAP simply selects which buffer is displayed on the screen; specifically, it flips between the two. SWAP only affects what is displayed on the screen; BUFFER only affects the target of the drawing commands.

I think that's it :).

```
*
* GRABAS
*
* A graphics extension for C-64 BASIC
*
* SLJ 12/29/96 (Completed 2/10/97)
* v1.0
*
         ORG $0801

* Constants

TXTPTR   = $7A              ;BASIC text pointer
IERROR   = $0300
ICRUNCH  = $0304            ;Crunch ASCII into token
IQPLOP   = $0306            ;List
IGONE    = $0308            ;Execute next BASIC token

CHRGET   = $73
CHRGOT   = $79
CHROUT   = $FFD2
GETBYT   = $B79E            ;BASIC routine
GETPAR   = $B7EB            ;Get a 16,8 pair of numbers
CHKCOM   = $AEFD
NEW      = $A642
CLR      = $A65E

LINNUM   = $14              ;Number returned by GETPAR

TEMP     = $FF
TEMP2    = $FB
```

```
POINT    = $FD
Y1       = $05
X1       = LINNUM
X2       = $02
Y2       = $04
DY       = $26
DX       = $27
BUF      = $0200           ;Input buffer

CHUNK1   = $69             ;Circle routine stuff
OLDCH1   = $6A
CHUNK2   = $6B
OLDCH2   = $6C
CX       = $A3
CY       = $A5
X        = $6D
Y        = $6E
RADIUS   = $6F
LCOL     = $A6             ;Left column
RCOL     = $A7
TROW     = $A8             ;Top row
BROW     = $A9             ;Bottom row



         DA :LINK          ;link
         DA 1997
         DFB $9E           ;SYS
         TXT '2063:'
         DFB $A2           ;NEW
         DFB 00            ;End of line
:LINK    DA 0              ;end of program

INSTALL  LDA #<PBEGIN
         STA POINT
         LDA #>PBEGIN
         STA POINT+1
         LDA #<PEND        ;Number of bytes to copy
         SEC
         SBC #<PBEGIN
         STA TEMP2
         LDA #>PEND
         SBC #>PBEGIN
         STA TEMP2+1
         LDA #$C0          ;Copy to $C000
         STA X2+1
         LDY #00
         STY X2
:LOOP    LDA (POINT),Y
         STA (X2),Y
         INY
         BNE :LOOP
         INC POINT+1
         INC X2+1
         DEC TEMP2+1
         BNE :LOOP
         LDY TEMP2
:LOOP2   LDA (POINT),Y
         STA (X2),Y
         DEY
         CPY #$FF
         BNE :LOOP2
         LDX #5            ;Copy CURRENT vectors
:LOOP3   LDA ICRUNCH,X
         STA OLDCRNCH,X
         DEX
         BPL :LOOP3
         JMP INIT

         TXT 'so, you want a secret message, eh? '
         TXT 'narnia, narnia, narnia, awake.'
         TXT ' love. think. speak. be walking trees.'
         TXT ' be talking beasts. be divine waters.'
         TXT 'stephen l. judd wuz here 1/20/97'

PBEGIN
         ORG $C000

*
* Init routine -- modify vectors
* and set up values.
```

```
*
INIT
         LDX #5            ;Copy vectors
:LOOP    LDA :TABLE,X
         STA ICRUNCH,X
         DEX
         BPL :LOOP
         INX
         STX ORGX
         STX ORGY

         JMP MODE17        ;Mode 17
:TEMP    DFB 05
:TABLE   DA CRUNCH
         DA LIST
         DA EXECUTE
JMPCRUN  DFB $4C           ;JMP
OLDCRNCH DS 2              ;Old CRUNCH vector
OLDLIST  DS 2
OLDEXEC  DS 2

*
* Keyword list
* Keywords are stored as normal text,
* followed by the token number.
* All tokens are >128,
* so they easily mark the end of the keyword
*

KEYWORDS
         TXT 'plot',E0
         TXT 'line',E1
         TXT 'circle',E2
         TXT 'gr',91,E3    ;grON
         TXT 'groff',E4    ;Graphics off
         TXT 'mode',E5
         DFB $B0           ;OR
         TXT 'igin',E6
         TXT 'clear',E7    ;Clear bitmap
         TXT 'buffer',E8   ;Set draw buffer
         TXT 'swap',E9     ;Swap foreground and background
         TXT 'col',B0,EA   ;Set color
         DFB 00            ;End of list

*
* Table of token locations-1
* Subtract $E0 first
* Then check to make sure number isn't greater than NUMWORDS
*
TOKENLOC
:E0      DA PLOT-1
:E1      DA LINE-1
:E2      DA CIRCLE-1
:E3      DA GRON-1
:E4      DA GROFF-1
:E5      DA MODE-1
:E6      DA ORIGIN-1
:E7      DA CLEAR-1
:E8      DA BUFFER-1
:E9      DA SWAP-1
:EA      DA COLOR-1
HITOKEN  EQU $EB

*
* CRUNCH -- If this is one of our keywords, then tokenize it
*
CRUNCH
         JSR JMPCRUN       ;First crunch line normally
         LDY #05           ;Offset for KERNAL
                           ;Y will contain line length+5
:LOOP    STY TEMP
         JSR ISWORD        ;Are we at a keyword?
         BCS :GOTCHA
:NEXT
         JSR NEXTCHAR
         BNE :LOOP         ;Null byte marks end
         STA BUF-3,Y       ;00 line number
         LDA #$FF          ;'tis what A should be
         RTS               ;Buh-bye
* Insert token and crunch line
:GOTCHA
```

```
         LDX TEMP          ;If so, A contains opcode
         STA BUF-5,X
:MOVE    INX
         LDA BUF-5,Y
         STA BUF-5,X       ;Move text backwards
         BEQ :NEXT
         INY
         BPL :MOVE

*
* ISWORD -- Checks to see if word is
* in table.  If a word is found, then
* C is set, Y is one past the last char
* and A contains opcode.  Otherwise,
* carry is clear.
*
* On entry, TEMP must contain current
* character position.
*
ISWORD
         LDX #00
:LOOP    LDY TEMP
:LOOP2   LDA KEYWORDS,X
         BEQ :NOTMINE
         CMP #$E0
         BCS :RTS          ;Tokens are >=$E0
         CMP BUF-5,Y
         BNE :NEXT
         INY               ;Success!  Go to next char
         INX
         BNE :LOOP2
:NEXT
         INX
         LDA KEYWORDS,X    ;Find next keyword
         CMP #$E0
         BCC :NEXT
         INX
         BNE :LOOP         ;And check again
:NOTMINE CLC
:RTS     RTS

*
* NEXTCHAR finds the next char
* in the buffer, skipping
* spaces and quotes.  On
* entry, TEMP contains the
* position of the last spot
* read.  On exit, Y contains
* the index to the next char,
* A contains that char, and Z is set if at end of line.
*
NEXTCHAR
         LDY TEMP
:LOOP    INY
         LDA BUF-5,Y
         BEQ :DONE
         CMP #$8F          ;REM
         BNE :CONT
         LDA #00
:SKIP    STA TEMP2         ;Find matching character
:LOOP2   INY
         LDA BUF-5,Y
         BEQ :DONE
         CMP TEMP2
         BNE :LOOP2        ;Skip to end of line
         BEQ :LOOP
:CONT
         CMP #$20          ;Space
         BEQ :LOOP
         CMP #$22          ;Quote
         BEQ :SKIP
:DONE    RTS

*
* LIST -- patches the LIST routine
* to list my tokens correctly.
*
LIST
         CMP #$E0
         BCC :NOTMINE      ;Not my token
         CMP #HITOKEN
         BCS :NOTMINE
         BIT $0F          ;Check for quote mode
         BMI :NOTMINE
         SEC
         SBC #$DF         ;Find the corresponding text
         TAX
         STY $49
         LDY #00
:LOOP    DEX
         BEQ :DONE
:LOOP2   INY
         LDA KEYWORDS,Y
         CMP #$E0
         BCC :LOOP2
         INY
         BNE :LOOP
:DONE    LDA KEYWORDS,Y
         BMI :OUT
         JSR $FFD2
         INY
         BNE :DONE
:OUT     CMP #$B0         ;OR
         BEQ :OR
         CMP #$E0         ;It might be BASIC token
         BCS :CONT        ;e.g. GRON
         LDY $49
:NOTMINE AND #$FF
         JMP (OLDLIST)    ;QPLOP
:CONT    LDY $49
         JMP $A700        ;Normal exit
:OR      LDA #'o'         ;For ORIGIN
         JSR CHROUT
         LDA #'r'
         JSR CHROUT
         INY
         BNE :DONE

*
* EXECUTE -- if this is one of my
* tokens, then execute it.
*
EXECUTE
         JSR CHRGET
         PHP
         CMP #$E0
         BCC :NOTMINE
         CMP #HITOKEN
         BCS :NOTMINE
         PLP
         JSR :DISP
         JMP $A7AE        ;Exit through NEWSTT
:DISP
         EOR #$E0
         ASL              ;Mult by two
         TAX
         LDA TOKENLOC+1,X
         PHA
         LDA TOKENLOC,X
         PHA
         JMP CHRGET       ;Exit to routine
:NOTMINE PLP
         JMP $A7E7        ;Normal routine

*
* PLOT -- plot a point!
*
ORGX     DFB 00           ;Upper-left corner of the screen
ORGY     DFB 00
DONTPLOT DFB 01           ;0=Don't plot point, just compute
                          ;coordinates (used by e.g. circles)

PLOT
         JSR GETPAR       ;Get coordinate pair
         LDA LINNUM       ;Add in origin offset
         SEC
         SBC ORGX
         STA LINNUM
         BCS :CONT1
         DEC LINNUM+1
         BMI :ERROR       ;Underflow
         SEC
```

```
:CONT1   TXA
         SBC ORGY
         BCC :ERROR
         TAX
         CPX #200          ;Check range
         BCS :ERROR
         LDA LINNUM
         CMP #<320
         LDA LINNUM+1
         SBC #>320
         BCC SETPOINT
:ERROR   RTS               ;Just don't plot point
*:ERROR LDX #14
* JMP (IERROR)
SETPOINT                   ;Alternative entry point
                           ;X=y-coord, LINNUM=x-coord
* ;X is preserved
* STX TEMP2
* STY TEMP2+1
                           ;On exit, X,Y are AND #$07
                           ;i.e. are set up correctly.
         TXA
         AND #248
         STA POINT
         LSR
         LSR
         LSR
         ADC BASE          ;Base of bitmap
         STA POINT+1
         LDA #00
         ASL POINT
         ROL
         ASL POINT
         ROL
         ASL POINT
         ROL
         ADC LINNUM+1
         ADC POINT+1
         STA POINT+1
         TXA
         AND #7
         TAY
         LDA LINNUM
         AND #248
         CLC               ;Overflow is possible!
         ADC POINT
         STA POINT
         BCC SETPIXEL
         INC POINT+1
SETPIXEL
         LDA LINNUM
         AND #$07
         TAX
         LDA DONTPLOT
         BEQ :RTS
         LDA POINT+1
         SEC
         SBC BASE          ;Overflow check
         CMP #$20
         BCS :RTS
         SEI               ;Get underneath ROM
         LDA #$34
         STA $01

         LDA (POINT),Y
         EOR BITMASK
         AND BITTAB,X
         EOR (POINT),Y
         STA (POINT),Y

         LDA #$37
         STA $01
         CLI
* LDX TEMP2
* LDY TEMP2+1
                           ;On exit, X,Y are AND #$07
                           ;i.e. are set up correctly.
                           ;for more plotting
:RTS     RTS

BITMASK  DFB #$FF          ;Set point
```

```
BITTAB   DFB $80,$40,$20,$10,$08,$04,$02,$01

*-------------------------------
* Drawin' a line.  A fahn lahn.
*
* To deal with off-screen coordinates, the current row
* and column (40x25) is kept track of.  These are set
* negative when the point is off the screen, and made
* positive when the point is within the visible screen.

* Little bit position table
BITCHUNK HEX FF7F3F1F0F070301
CHUNK    EQU X2
OLDCHUNK EQU X2+1

* DOTTED -- Set to $01 if doing dotted draws (diligently)
* X1,X2 etc. are set up above (x2=LINNUM in particular)
* Format is LINE x2,y2,x1,y1

LINE
         JSR GETPAR
         STX Y2
         LDA LINNUM
         STA X2
         LDA LINNUM+1
         STA X2+1
         JSR CHKCOM
         JSR GETPAR
         STX Y1

:CHECK   LDA X2            ;Make sure x1<x2
         SEC
         SBC X1
         TAX
         LDA X2+1
         SBC X1+1
         BCS :CONT
         LDA Y2            ;If not, swap P1 and P2
         LDY Y1
         STA Y1
         STY Y2
         LDA X1
         LDY X2
         STY X1
         STA X2
         LDA X2+1
         LDY X1+1
         STA X1+1
         STY X2+1
         BCC :CHECK

:CONT    STA DX+1
         STX DX

         LDX #$C8          ;INY
         LDA Y2            ;Calculate dy
         SEC
         SBC Y1
         BCS :DYPOS        ;Is y2>=y1?
         EOR #$FF          ;Otherwise dy=y1-y2
         ADC #$01
         LDX #$88          ;DEY

:DYPOS   STA DY
         STX YINCDEC
         STX XINCDEC

         LDA X1            ;Sub origin from 1st point
         SEC
         SBC ORGX
         STA X1
         LDA X1+1
         SBC #00
         STA X1+1
         PHP               ;Save carry flag
         STA TEMP          ;Next compute column
         LDA X1
         LSR TEMP
         ROR
         LSR TEMP
         ROR
         LSR TEMP
         ROR
         STA CX            ;X-column
         PLP
         BCC :NEGX         ;If negative, then fix up
         CMP #40           ;If past column 40, then punt!
         BCC :CONT1
         RTS
:NEGX    LDA X1            ;coordinate start and count
         AND #$07
         STA X1
         LDA #00
         STA X1+1
         LDA CX

:CONT1   LDA Y1            ;Now do the same for Y
         SEC
         SBC ORGY
         STA Y1
         TAX               ;X=y-coord
         PHP               ;Save carry bit
         LSR
         LSR
         LSR
         STA CY            ;Y-column (well, OK, row then)
         PLP
         BCC :NEGY         ;If negative, then fix stuff up!
         SBC #25           ;Check if we are past bottom of
         BCC :CONT2        ;screen
         ORA #$80          ;Otherwise, 128+rows past 24
         STA CY            ;(for plot range checking)
         TXA
         AND #$07
         ORA #8*24         ;Start in last row
         TAX
         BMI :CONT2
:NEGY    ORA #$E0          ;Set high bits of column
         STA CY
         TXA
         AND #$07
         TAX               ;Start in 1st row
:CONT2

         LDA #00
         STA DONTPLOT
         JSR SETPOINT      ;Set up X,Y and POINT
         INC DONTPLOT
         LDA BITCHUNK,X
         STA OLDCHUNK
         STA CHUNK

         SEI               ;Get underneath ROM
         LDA #$34
         STA $01

         LDX DY
         CPX DX            ;Who's bigger: dy or dx?
         BCC STEPINX       ;If dx, then...
         LDA DX+1
         BNE STEPINX

*
* Big steps in Y
*
*   To simplify my life, just use PLOT to plot points.
*
*   No more!
*   Added special plotting routine -- cool!
*
*   X is now counter, Y is y-coordinate
*
* On entry, X=DY=number of loop iterations, and Y=
*   Y1 AND #$07
STEPINY
         LDA #00
         STA OLDCHUNK      ;So plotting routine will work right
         LDA CHUNK
         SEC
         LSR               ;Strip the bit
         EOR CHUNK
         STA CHUNK
         TXA
         BNE :CONT        ;If dy=0 it's just a point
         INX
:CONT    LSR              ;Init counter to dy/2
*
* Main loop
*
YLOOP    STA TEMP
* JSR LINEPLOT

         LDA CX           ;Range check
         ORA CY
         BMI :SKIP

         LDA (POINT),Y    ;Otherwise plot
         EOR BITMASK
         AND CHUNK
         EOR (POINT),Y
         STA (POINT),Y
:SKIP
YINCDEC  INY              ;Advance Y coordinate
         CPY #8
         BCC :CONT        ;No prob if Y=0..7
         JSR FIXY
:CONT    LDA TEMP         ;Restore A
         SEC
         SBC DX
         BCC YFIXX
YCONT    DEX              ;X is counter
         BNE YLOOP
YCONT2   LDA (POINT),Y    ;Plot endpoint
         EOR BITMASK
         AND CHUNK
         EOR (POINT),Y
         STA (POINT),Y
YDONE
         LDA #$37
         STA $01
         CLI
         RTS

YFIXX                     ;x=x+1
         ADC DY
         LSR CHUNK
         BNE YCONT        ;If we pass a column boundary...
         ROR CHUNK        ;then reset CHUNK to $80
         STA TEMP2
         LDA CX
         BMI :CONT        ;Skip if column is negative
         CMP #39          ;End if move past end of screen
         BCS YDONE

         LDA POINT        ;And add 8 to POINT
         ADC #8
         STA POINT
         BCC :CONT
         INC POINT+1
:CONT    INC CX           ;Increment column
         LDA TEMP2
         DEX
         BNE YLOOP
         BEQ YCONT2

*
* Big steps in X direction
*
* On entry, X=DY=number of loop iterations, and Y=
*   Y1 AND #$07

COUNTHI  DFB 00           ;Temporary counter
                          ;only used once
STEPINX
         LDX DX
         LDA DX+1
         STA COUNTHI
         LSR              ;Need bit for initialization
         STA Y1           ;High byte of counter
         TXA
         BNE :CONT        ;Could be $100
         DEC COUNTHI
:CONT    ROR
```

```
*
* Main loop
*
XLOOP
         LSR CHUNK
         BEQ XFIXC         ;If we pass a column boundary...
XCONT1   SBC DY
         BCC XFIXY         ;Time to step in Y?
XCONT2   DEX
         BNE XLOOP
         DEC COUNTHI       ;High bits set?
         BPL XLOOP
XDONE
         LSR CHUNK         ;Advance to last point
         JSR LINEPLOT      ;Plot the last chunk
EXIT     LDA #$37
         STA $01
         CLI
         RTS
*
* CHUNK has passed a column, so plot and increment pointer
* and fix up CHUNK, OLDCHUNK.
*
XFIXC
         STA TEMP
         JSR LINEPLOT
         LDA #$FF
         STA CHUNK
         STA OLDCHUNK
         LDA CX
         BMI :CONT         ;Skip if column is negative
         CMP #39           ;End if move past end of screen
         BCS EXIT

         LDA POINT
         ADC #8
         STA POINT
         BCC :CONT
         INC POINT+1
:CONT    INC CX
         LDA TEMP
         JMP XCONT1
*
* Check to make sure there isn't a high bit, plot chunk,
* and update Y-coordinate.
*
XFIXY
         DEC Y1            ;Maybe high bit set
         BPL XCONT2
         ADC DX
         STA TEMP
         LDA DX+1
         ADC #$FF          ;Hi byte
         STA Y1

         JSR LINEPLOT      ;Plot chunk
         LDA CHUNK
         STA OLDCHUNK

         LDA TEMP
XINCDEC  INY               ;Y-coord
         CPY #8            ;0..7 is ok
         BCC XCONT2
         STA TEMP
         JSR FIXY
         LDA TEMP
         JMP XCONT2

*
* Subroutine to plot chunks/points (to save a little
* room, gray hair, etc.)
*
LINEPLOT                   ;Plot the line chunk

         LDA CX
         ORA CY
         BMI :SKIP

         LDA (POINT),Y     ;Otherwise plot
         EOR BITMASK
         ORA CHUNK
         AND OLDCHUNK
         EOR CHUNK
         EOR (POINT),Y
         STA (POINT),Y
:SKIP
         RTS

*
* Subroutine to fix up pointer when Y decreases through
* zero or increases through 7.
*
FIXY     CPY #255          ;Y=255 or Y=8
         BEQ :DECPTR
:INCPTR                    ;Add 320 to pointer
         LDY #0            ;Y increased through 7
         LDA CY
         BMI :CONT1        ;If negative, then don't update
         CMP #24
         BCS :TOAST        ;If at bottom of screen then quit
         LDA POINT
         ADC #<320
         STA POINT
         LDA POINT+1
         ADC #>320
         STA POINT+1
:CONT1   INC CY
         RTS
:DECPTR                    ;Okay, subtract 320 then
         LDY #7            ;Y decreased through 0
         LDA CY
         BEQ :TOAST
         BMI :CONT2
         CMP #$7F          ;It is possible we just decreased
         BNE :C1           ;through row 25
         LDA #24
         STA CY            ;In which case, set correct row
:C1      LDA POINT
         SEC
         SBC #<320
         STA POINT
         LDA POINT+1
         SBC #>320
         STA POINT+1
:CONT2   DEC CY
         RTS
:TOAST   PLA               ;Remove old return address
         PLA
         JMP EXIT          ;Restore interrupts, etc.

*
* CIRCLE draws a circle of course, using my
* super-sneaky algorithm.
*   CIRCLE cx,cy,radius (16,8,8)
*

CIRCLE
         JSR GETPAR
         STX CY            ;CX,CY = center

         LDA X1
         SEC
         SBC ORGX
         STA CX
         STA X1
         LDA X1+1
         SBC #00
         STA CX+1
         STA X1+1
         PHP               ;Save carry
         LSR               ;Compute which column we start
         LDA CX            ;in
         ROR
         LSR
         LSR
         PLP
         BCS :CONT         ;Underflow means negative column
         TAX
         LDA X1            ;Set X to first column
         AND #$07
         STA X1
         LDA #00
         STA X1+1
         TXA
         ORA #$E0          ;so set high bits
:CONT    STA RCOL
         STA LCOL
         BMI :SKIP
         CMP #40           ;Check for benefit of SETPOINT
         BCC :SKIP
         LDA X1            ;Set X in last column
         AND #$07
         ORA #64-8         ;312+X AND 7
         STA X1
         LDA #1
         STA X1+1
:SKIP
         JSR CHKCOM
         JSR GETBYT
CIRCENT                    ;Alternative entry point
         STX Y
         STX RADIUS
         TXA
         BNE :C            ;Skip R=0
         LDX CY
         JMP SETPOINT      ;Plot it as a point.
:C       CLC
         ADC CY
         BCS :BLAH
         SEC
         SBC ORGY
         BCS :C4           ;cy+y<orgy implies circle off screen
:RTS     RTS

:BLAH    SBC ORGY          ;Always positive
         BCS :C3           ;Handle overflow sneaky
:C4      TAX
         CMP #200          ;If Y>200 then set pointer to
         BCC :C2           ;last row, but set TROW
         CLC               ;correctly
:C3      TAY
         AND #$07
         ORA #$C0          ;Last row, set Y1 correctly
         TAX
         TYA
:C2      ROR
         LSR
         LSR
         STA TROW          ;Top row

         LDA #00
         STA DONTPLOT      ;Don't plot points
         JSR SETPOINT      ;Plot XC,YC+Y
         STY Y2            ;Y AND 07
         LDA BITCHUNK,X
         STA CHUNK1        ;Forwards chunk
         STA OLDCH1
         LSR
         EOR #$FF
         STA CHUNK2        ;Backwards chunk
         STA OLDCH2
         LDA POINT
         STA TEMP2         ;TEMP2 = forwards high pointer
         STA X2            ;X2 = backwards high pointer
         LDA POINT+1
         STA TEMP2+1
         STA X2+1

         LDA CY            ;Now compute upper points
         SEC
         SBC ORGY
         BCS :CSET
         SEC               ;We are so very negative
         SBC Y
         CLC
         BCC :BNEG
:CSET    SBC Y             ;Compute CY-Y-ORGY
:BNEG    PHP
         TAX
         LSR               ;Compute row
         LSR
         LSR
         STA BROW
         PLP
         BCS :CONT
         ORA #$E0          ;Make row negative
         STA BROW
         TXA
         AND #07           ;Handle underflow special!
         TAX
:CONT    JSR SETPOINT      ;Compute new coords
         STY Y1
         LDA POINT
         STA X1            ;X1 will be the backwards
         LDA POINT+1       ;low-pointer
         STA X1+1          ;POINT will be forwards

         SEI               ;Get underneath ROM
         LDA #$34
         STA $01

         LDA Y
         LSR               ;A=r/2
         LDX #00
         STX X             ;y=0

* Main loop

:LOOP
         INC X             ;x=x+1

         LSR CHUNK1        ;Right chunk
         BNE :CONT1
         JSR UPCHUNK1      ;Update if we move past a column
:CONT1   ASL CHUNK2
         BNE :CONT2
         JSR UPCHUNK2
:CONT2                     ;LDA TEMP
         SEC
         SBC X             ;a=a-x
         BCS :LOOP

         ADC Y             ;if a<0 then a=a+y; y=y-1
         TAX
         JSR PCHUNK1
         JSR PCHUNK2
         LDA CHUNK1
         STA OLDCH1
         LDA CHUNK2
         STA OLDCH2
         TXA

         DEC Y             ;(y=y-1)

         DEC Y2            ;Decrement y-offest for upper
         BPL :CONT3        ;points
         JSR DECYOFF
:CONT3   LDY Y1
         INY
         STY Y1
         CPY #8
         BCC :CONT4
         JSR INCYOFF
:CONT4
         LDY X
         CPY Y             ;if y<=x then punt
         BCC :LOOP         ;Now draw the other half
*
* Draw the other half of the circle by exactly reversing
* the above!
*
NEXTHALF
         LSR OLDCH1        ;Only plot a bit at a time
         ASL OLDCH2
         LDA RADIUS        ;A=-R/2-1
         LSR
         EOR #$FF
:LOOP
         TAX
         JSR PCHUNK1       ;Plot points
         JSR PCHUNK2
         TXA
         DEC Y2            ;Y2=bottom
         BPL :CONT1
         JSR DECYOFF
:CONT1   INC Y1
         LDY Y1
         CPY #8
         BCC :CONT2
         JSR INCYOFF
:CONT2
         LDX Y
         BEQ :DONE
         CLC
         ADC Y             ;a=a+y
         DEC Y             ;y=y-1
         BCC :LOOP

         INC X
         SBC X             ;if a<0 then x=x+1; a=a+x
         LSR CHUNK1
         BNE :CONT3
         TAX
         JSR UPCH1         ;Upchunk, but no plot
:CONT3   LSR OLDCH1        ;Only the bits...
         ASL CHUNK2        ;Fix chunks
         BNE :CONT4
         TAX
         JSR UPCH2
:CONT4   ASL OLDCH2
         BCS :LOOP
:DONE
CIRCEXIT                   ;Restore interrupts
         LDA #$37
         STA $01
         CLI
         LDA #1            ;Re-enable plotting
         STA DONTPLOT
         RTS
*
* Decrement upper pointers
*
DECYOFF
         TAY
         LDA #7
         STA Y2
         LDA TROW          ;First check to see if Y is in
         BEQ EXIT2
         CMP #25           ;range (rows 0-24)
         BCS :SKIP
         LDA X2            ;If we pass through zero, then
         SEC
         SBC #<320         ;subtract 320
         STA X2
         LDA X2+1
         SBC #>320
         STA X2+1
         LDA TEMP2
         SEC
         SBC #<320
         STA TEMP2
         LDA TEMP2+1
         SBC #>320
         STA TEMP2+1
:SKIP    TYA
         DEC TROW
         RTS
EXIT2    PLA               ;Grab return address
         PLA
         JMP CIRCEXIT      ;Restore interrupts, etc.

* Increment lower pointers
INCYOFF
         TAY
         LDA #00
         STA Y1
         LDA BROW
         BMI :ISKIP        ;If <0 then don't update pointer.
         CMP #24           ;If we hit bottom of screen then
         BEQ EXIT2         ;just quit
         LDA X1
         CLC
         ADC #<320
         STA X1
         LDA X1+1
         ADC #>320
         STA X1+1
         LDA POINT
         CLC
         ADC #<320
         STA POINT
         LDA POINT+1
         ADC #>320
         STA POINT+1
:ISKIP   TYA
         INC BROW
         RTS

*
* UPCHUNK1 -- Update right-moving chunk pointers
*             Due to passing through a column
*
UPCHUNK1
         TAX
         JSR PCHUNK1
UPCH1    LDA #$FF          ;Alternative entry point
         STA CHUNK1
         STA OLDCH1
         LDA RCOL
         BMI :DONE         ;Can start negative
         LDA TEMP2
         CLC
         ADC #8
         STA TEMP2
         BCC :CONT
         INC TEMP2+1
         CLC
:CONT    LDA POINT
         ADC #8
         STA POINT
         BCC :DONE
         INC POINT+1
:DONE    TXA
         INC RCOL
         RTS

*
* UPCHUNK2 -- Update left-moving chunk pointers
*
UPCHUNK2
         TAX
         JSR PCHUNK2
UPCH2    LDA #$FF
         STA CHUNK2
         STA OLDCH2
         LDA LCOL
         CMP #40
         BCS :DONE
         LDA X2
         SEC
         SBC #8
         STA X2
         BCS :CONT
         DEC X2+1
         SEC
:CONT    LDA X1
         SBC #8
         STA X1
         BCS :DONE
         DEC X1+1
:DONE    TXA
         DEC LCOL
         RTS
*
* Plot right-moving chunk pairs for circle routine
*
PCHUNK1

         LDA RCOL          ;Make sure we're in range
         CMP #40
         BCS :SKIP2
         LDA CHUNK1        ;Otherwise plot
         EOR OLDCH1
         STA TEMP
         LDA BROW          ;Check for underflow
         BMI :SKIP
         LDY Y1
         LDA (POINT),Y
         EOR BITMASK
         AND TEMP
         EOR (POINT),Y
         STA (POINT),Y

:SKIP    LDA TROW          ;If CY+Y >= 200...
         CMP #25
         BCS :SKIP2
         LDY Y2
         LDA (TEMP2),Y
         EOR BITMASK
         AND TEMP
         EOR (TEMP2),Y
         STA (TEMP2),Y
:SKIP2
         RTS

*
* Plot left-moving chunk pairs for circle routine
*
PCHUNK2

         LDA LCOL          ;Range check in X
         CMP #40
         BCS :SKIP2
         LDA CHUNK2        ;Otherwise plot
         EOR OLDCH2
         STA TEMP
         LDA BROW          ;Check for underflow
         BMI :SKIP
         LDY Y1
         LDA (X1),Y
         EOR BITMASK
         AND TEMP
         EOR (X1),Y
         STA (X1),Y

:SKIP    LDA TROW          ;If CY+Y >= 200...
         CMP #25
         BCS :SKIP2
         LDY Y2
         LDA (X2),Y
         EOR BITMASK
         AND TEMP
         EOR (X2),Y
         STA (X2),Y
:SKIP2
         RTS

*
* GRON -- turn graphics on.  If a number appears
* afterwards, then initialize the colormap to that
* number and clear the bitmap.
*
BASE     DFB $E0           ;Address of bitmap, hi byte
BANK     DFB 0             ;Bank 3=default
OLDBANK  DFB $FF           ;VIC old bank
OLDD018  DFB 00

GRON
         LDA $D011         ;Skip if bitmap is already on.
         AND #$20
         BNE CLEAR
         LDA $DD02         ;Set the data direction regs
         ORA #3
         STA $DD02
         LDA $DD00
         PHA
         AND #$03
         STA OLDBANK
         PLA
         AND #252
         ORA BANK
         STA $DD00

         LDA $D018
         STA OLDD018
         LDA #$38          ;Set color map to base+$1C00
         STA $D018         ;bitmap to 2nd 8k
         LDA $D011        ;And turn on bitmap
         ORA #$20
         STA $D011

CLEAR    JSR CHRGOT       ;See if there's a color
         BEQ GRONDONE
         JSR GETBYT       ;Get the char
CLEARCOL LDA #00          ;Low byte of base address
         STA POINT
         LDA BASE         ;Colormap is at base-$14
         SEC
         SBC #$14
         STA POINT+1
         TXA
         LDY #00
         LDX #4
:LOOP    STA (POINT),Y
         INY
         BNE :LOOP
         INC POINT+1
         DEX
         BNE :LOOP
         LDA BASE         ;Now clear bitmap
         STA POINT+1
         LDX #32
         TYA
:LOOP2   STA (POINT),Y
         INY
         BNE :LOOP2
         INC POINT+1
         DEX
         BNE :LOOP2

GRONDONE RTS

* GROFF -- Restore old values if graphics are on.
GROFF
         LDA $D011
         AND #$20
         BEQ GDONE
GSET     LDA $DD02        ;Set the data direction regs
         ORA #3
         STA $DD02
         LDA $DD00
         AND #$7C
         ORA OLDBANK
         STA $DD00

         LDA OLDD018
         STA $D018

         LDA $D011
         AND #$FF-$20
         STA $D011
GDONE    RTS

*
* COLOR -- Set drawing color
*
COLOR
         JSR GETBYT
COLENT   CPX #00          ;MODE enters here
         BEQ :C2
:C1      CPX #01
         BNE :RTS
         LDX #$FF
:C2      STX BITMASK
:RTS     RTS

*
* MODE -- catch-all command.  Currently implemented:
*   00  Erase (background color)
*   01  Foreground color
*   16  SuperCPU mode -- screen -> A000, etc.
*   17  Normal mode
*   18  Double buffer mode
*
*  Anything else -> BITMASK
*
MODENUM  DFB 17           ;Current mode
```

```
MODE
         JSR GETBYT
         CPX #2
         BCC COLENT
:C16     CPX #16
         BNE :C18
         STX MODENUM
:SET16   LDA #$A0          ;Bitmap -> $A000
         STA BASE
         LDA #01
         STA BANK          ;Bank 2
         STA OLDBANK
         LDA #$FF          ;End of BASIC memory
         STA $37
         STA $33
         LDA #$87
         STA $38
         STA $34
         LDA #$24          ;Screen mem -> $8800
         STA OLDD018
         JSR GSET          ;Part of GROFF
         LDA #$88
         STA 648           ;Tell BASIC where the screen is
         STA $D07E         ;Enable SuperCPU regs
         STA $D074         ;Bank 2 optimization
         STA $D07F         ;Disable regs
         RTS
:C18     CPX #18           ;Double-buffer mode!
         BNE :C17
         STX MODENUM
         JSR :SET16        ;Set up mode 16
         STA $D07E
         STA $D077         ;Turn off optimization
         STA $D07F
         RTS
:C17     CPX #17
         BNE MODEDONE
MODE17   STX MODENUM
         LDA #$E0
         STA BASE
         LDA #00           ;Bank 3
         STA BANK
         LDA #3            ;Bank 0 == normal bank
         STA OLDBANK
         LDA #$FF
         STA $37
         STA $33
         LDA #$9F
         STA $38
         STA $34
         LDA #$14          ;Screen mem -> $0400
         STA OLDD018
         JSR GSET          ;Part of GROFF
         LDA #$04
         STA 648           ;Tell BASIC where the screen is
         STA $D07E
         STA $D077         ;No optimization
         STA $D07F
         RTS
MODEDONE STX BITMASK
         RTS


*
* BUFFER -- Sets the current drawing buffer to 1 or 2,
*   depending on arg being even or odd.  If double-
*   buffer mode is not enabled then punt.
*
*   Now, buffer=0 swaps draw buffers, even/odd otherwise.
*
BUFFER
         JSR GETBYT
         LDA MODENUM
         CMP #18
         BNE :PUNT
         LDY #$A0
         TXA
         BNE :CONT
         CPY BASE
         BNE :CONT
         LDA #1
```

```
:CONT    LSR
         BCC :LOW         ;even = low buffer
         LDY #$E0         ;odd = high buffer
:LOW     STY BASE
:PUNT    RTS

*
* SWAP -- Swap displayed buffers.  MODE 18 must
*   be enabled first.
*
SWAP
         LDA MODENUM
         CMP #18
         BNE :PUNT
         LDA $DD00        ;Ooooooohhh, real tough!
         EOR #$01
         STA $DD00
:PUNT    RTS

*
* ORIGIN -- Set upper-left corner of the screen to
*   new coordinate offset.
*
ORIGIN
         JSR GETBYT
         STX ORGX
         JSR CHKCOM
         JSR GETBYT
         STX ORGY
         RTS

         ORG              ;re-org
PEND                      ;To get that label right :)
```

--
The BLARG distribution binary is available from 'The Fridge', Mr. Judd's archival web page on the internet at http://stratus.esam.nwu.edu/judd/fridge

```
::::::::::d:i:s:C=:o:v:e:r:y::::::::::::::::i:s:s:u:e::3::::::::::::::::::::
/S02::$d000:::::::::::::::::::S O F T W A R E::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

A CLOSER LOOK AT THE VIC II'S OUTPUT

by Adrian Gonzalez (DW/Style) & George Taylor (Repose/Style)

```
------------------------------------------------------------------------------
         1         2         3         4         5         6         7
123456789012345678901234567890123456789012345678901234567890123456789012345678
```

Contents

0. Preface

This text was originally formatted to 78 columns for ease of use. This ensures compatibility with MSDOS "edit" with scrollbars, as well as 80 column readers that cause double spacing if the 81st character is a carriage return. The text is written in conversation style to ease the problem of dual authorship. Subsections not marked may be considered generic text and may have been written by either of us. You will not be able to view the ASCII diagrams in 40 columns. Major headings are marked with ---'s.

1. Introduction

DW:

While working on an image conversion project, I stumbled upon the need to find RGB values for the c64's colors. My first idea was using two monitors and matching the colors "by eye", however, this turned out to be more difficult than I expected. After trying other methods I decided the next logical step was to analyze the c64's output and determine the RGB values from there. Simple enough, eh? The only problem was that I had no clue as to what the VIC-II's output looked like. After digging up some books on TV theory, I decided to embark on a quest to find RGB values for the c64's colors, and I found many interesting things along the way.

This article is the result of many hours of research, programming, and taking measurements. I hope you find it interesting and perhaps even learn something new from it (!).

Repose:

Coincidently, I was working on exactly the same kind of image conversion project. So I got together with DW to discuss our common problem of finding the RGB colors of the 64. I also put out a request to comp.sys.cbm and was sent measurements from several people, including an excellent effort from Marko Makela.

1.1 Target audience

This article should be very interesting for people doing emulators, image converters, and other similar projects. Due to the rather technical nature of its content, it is not meant for beginners, however, the end result could be very useful for anybody interested in using other platforms to do graphics work for the c64. If you're not sure whether this article is for you or not, here are some terms you should be familiar with: scanline, frame, refresh rate, CRT, RGB, vertical and horizontal blanking. These terms are used all throughout the article, and it is assumed you already have a basic knowledge of how a TV works.

1.2 The legal stuff

Part of this article deals with taking measurements from your c64. If you decide to tinker with your c64 and something goes wrong, it's your fault, not ours. The authors will not be held responsible for damaged equipment, data loss, loss of sleep, loss of sanity, etc. Now, on with the show.

2. The NTSC Y/C video signal

There's basically three ways you can connect your c64 to a TV or monitor and get a color picture. The first and perhaps the most common is to use the RF modulated output with a TV tuned to channel 3 or 4. The second and third use the Video/Audio connector and require either a monitor or a TV/VCR with A/V inputs. We will get to each of these later on, but first, a little bit of history.

2.1 A little history.

DW:

In 1953, the National Television Systems Committee (NTSC) developed a standard that allowed the transmission of color images while remaining compatible with the large amount of black and white TV sets in widespread use at the time.   In the US, public broadcasting (using the color NTSC system) began in 1954. The same system was adopted in Japan, where it came into service in 1960. Other countries favored modifications of the NTSC system, such as PAL (Phase Alternating Line) and SECAM (Systeme Electronique Couleur Avec Memoire).

Repose:

PAL is used in many western european countries, and has technical advantages to NTSC. SECAM is used in eastern and middle eastern europe. It is only a semi standard in a way because video production is always done in PAL format, and only converted to SECAM for final transmission. The main point of this was information control, as it offers no advantages over PAL. As far as I know, VIC IIs were only made to conform to PAL or NTSC standards.

2.2 A closer look at black and white TV.

In designing a TV system, the engineers had to make several considerations. One had to do with bandwidth, which is the space that a TV channel takes in the radio frequency spectrum. To allow for many channels, and also to be easy for the ancient technology of that time, it was decided to split up the picture into two parts, and send each half sequentially.

The display on your TV is made up of a several hundred scanlines, composed of two fields which are interlaced to form a complete display, called a frame. This interlacing doesn't happen on the c64, but we will get to that later. First let's look at the fields:

```
Odd field                                 Even field

                                     Scanline     1 ++++++++++++++++++++++
Scanline   2 ----------------------
                                     Scanline     3 ++++++++++++++++++++++
Scanline   4 ----------------------
                                     Scanline     5 ++++++++++++++++++++++
Scanline   6 ----------------------      .
.                                        .
.
                                     Scanline 2*n-1 ++++++++++++++++++++++
Scanline 2*n ----------------------
                                     Scanline 2*n+1 ++++++++++++++++++++++
```

Each of the fields is 262 1/2 lines long (NTSC), (312 1/2 PAL) which means each frame is 525 (625 PAL) lines long. Your TV displays odd fields and even fields one after another, and thanks to what is called "persistence of vision" we see something like:

```
Scanline     1 ++++++++++++++++++++++
Scanline     2 -----------------------
Scanline     3 ++++++++++++++++++++++
Scanline     4 -----------------------
Scanline     6 ++++++++++++++++++++++
.
.
```

The second consideration the engineers had was how to represent a 2d image as a 1d voltage. To do this, they needed markers to separate a horizontal line and also odd and even fields.

So, let's take a closer look at what the voltage waveform for black and white scanlines looks like:

```
^  Voltage
|
|             Scanline n                                  Scanline n+2
|          __________________________________          ________________..
|         |   White level                    |         |
|         |                                  |         |
|         |                                  |         |
|         |                                  |         |
|         |                                  |         |
|___  ____|   Black level                    |___  ____|
|  |  |                                         |  |
|  |__| <- Horizontal sync pulses ------------> |__|
|
--------------------------------------------------------------------------->
Time
```

The scanline in this example is a simple white line. If you were to feed enough of these to your TV plus some vertical sync signals, you would get a white screen. The horizontal sync pulses tell the TV receiver when a scanline starts. They make up 25% of the total height of the signal.

The brightness at a particular point of the scanline is defined by the voltage of the waveform at that instant. With this in mind, if we wanted to create a display with a simple white vertical line at the middle of the screen, the waveform would look like this:

```
^  Voltage
|
|                   Scanline n                              Scanline n+2
|                        __                                            __
|           White level |  |                                          |  |
|                       |  |                                          |  |
|                        |  |                                         |  |
|           Black level  |  |                                         |  |
|___  __________________|  |____________________  __________________|  |
|  |  |                                          |  |
|  |__| <- Horizontal sync pulses ------------> |__|
|
---------------------------------------------------------------------------->
Time
```

Let us now turn our attention to the visible display area on your TV:

```
___                                                            ___
 ^        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX   6% of frame
 |        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ___ Vert. blank
 |        XXXXXXXX                                             ^
 |        XXXXXXXX                                             |
 |        XXXXXXXX                                             |
 |        XXXXXXXX                                             |
 |  Field  XXXXXXXX                                            |
 |  period XXXXXXXX                                            |
 |  (60hz) XXXXXXXX                                            |
 |        XXXXXXXX                                             |
 |        XXXXXXXX                                             |
 |        XXXXXXXX                                             |
 |        XXXXXXXX                                             |
 |        XXXXXXXX                                             |
 |        XXXXXXXX                                             |
 |        XXXXXXXX                                             |
_v_       XXXXXXXX                                            _v_
          |<---->|<------------------------------------------>|
           17% of                 83% of line
           line                   is visible
           Horiz.
           blank
          |<----------------------------------------------------->|
                      Line period: 63.5 microseconds
```

The X's in the graph represent areas in which the in which your screen is not visible. At the top we have the vertical blanking interval, and at the left we have the horizontal blanking interval. Please note that whether these intervals occur at the top or bottom, left or right or both, is not relevant. It is shown this way in the diagram for the sake of clarity. So, with that out of the way, let's see what happens in the vertical blank interval (vblank from now on):

```
^ Voltage
|
|    last                                                          time   ---->
|    scanline
|  __ _______ ___ ___ ___ ___ ___ __                            _ ___ ___ _..
|    |       |   |   |   |   |   |  |____|___|___|___|___|___|_| | |   |   |
     ^       ^                                          ^            ^
Horizontal   End of even field        Serrated vertical sync    6 equalizing
sync pulse   6 equalizing pulses      pulse                     pulses
```

Please note that the time scale has been compressed to be able to cover most of the vblank interval. Basically the vblank interval is composed of 6 equalizing pulses, one serrated vertical sync pulse, and 6 more equalizing pulses. The equalizing pulses and the serrations in the vertical sync pulse are spaced half a scanline apart. An interesting thing happens if we take away the equalizing pulses and serrations:

```
^ Voltage
|
|    last                                                          time   ---->
|    scanline
|  ________________________________                            ___________..
|                                   |                          |
|                                   |__________________________|

                                        Vertical sync pulse
```

We end up with a vertical sync pulse that looks very much like the horizontal sync pulses, except it runs at a much lower frequency. In NTSC, the field frequency is 60 Hz, and the frame frequency is 30hz (PAL has a 50Hz field rate and 25Hz frame rate). In other words, we get one of these vertical pulses on every field, to let the TV know when the field starts (just like the horizontal sync pulses tell it when a line starts). According to an old book on TV theory, "serrations are placed in the vertical sync pulse to stabilize the operation of the horizontal scanning generator during vertical retrace time".

One last thing remains before we move on to color television: interlacing. Interlacing is achieved by making the field size 262 1/2 lines long, which also changes the spacing between the equalizing pulses and the vertical sync pulse. It will not be discussed any further, though, because the c-64's output is not interlaced. Instead, it is composed of always fields, thus there are visible spaces between the scanlines. The field rate may be considered also the frame rate, since one field is a complete image in itself.

Note: it is not known to us if the c64 produces even or odd fields.

2.3 Let's talk about color.

Up until now, we've only talked about black and white TV signals, but if you remember the little bit of history in section 2.1, you know that color has to be introduced in a way that doesn't interfere with the BW signal. The VIC-II really outputs two signals (NTSC version only): the Luminance (Y) signal and the Chrominance (C) signal. The Y signal contains the brightness information of the output, while the C signal contains color information. The Y signal is just like the one described in the previous section, except it is not interlaced. The c64 has circuitry to mix the Y and C signals into a composite video signal which has both BW and color info. It then goes to an RF modulator which allows these signals (plus audio) to be viewed on a TV tuned to channel 3 or 4 (in the North American TV frequency).

Before we move on to describe the color signal, we will talk briefly about RGB. RGB stands for Red Green Blue, or in other words, a set of three primary colors, which are mixed in different amounts in order to obtain a broad gammut (range) of colors. An interesting thing to note is that it is impossible to reproduce the entire spectrum of visible light with a set of three (or any finite number) of real primary colors, however, RGB and other color systems do a fair job of being able to represent typical images.
RGB color is probably quite familiar to you, as it is the basis of the video circuitry in most computers and their monitors. However, NTSC/PAL signals use an encoding called YUV. Y is the brightness information, while UV are the C or color information. U and V are components of a vector. You can picture this vector as originating from the centre of a circle. It's angle is the tint of the color, and it's radius is the saturation of the color. The angle represents a full sprectrum, from red, to orange, yellow, green, blue, purple, back to red again. A highly satured color will be pure, while no saturation would give a grey. There are usually controls related to these properties on a TV, called tint and color, or hue and saturation.

2.4 The color signal

Color was added to B/W NTSC in a very clever way. It was added as a carrier on top of the normal signal. It is basically a sinewave of 3.574545Mhz (NTSC) (4.43Mhz PAL) with a varying phase and amplitude. To a B/W TV, this will be seen as a fine wavey detail in the luminance. But, a color TV uses a filter to separate the high frequencies as the color signal, and pass on the lower frequencies as the normal B/W signal. This separation is not perfect, and can cause several strange effects on the picture, known as "crawling dots", "hanging dots", and "color moire patterns" (NTSC, similar effects may appear in PAL). That is why, coincidently, the S-VHS format was invented, to physically separate the color signal from the luma signal. One question, that may have occurred to you, is how do you measure phase without a reference point? Fortunately, at the start of every rasterline there is a calibration color signal known as the color burst, and from this sinewave, the phase of the color signal is measured.

The sinewave then continues, at the same time as the luminence portion of rasterline is sent, both signals varying to specify a color and luminance.

```
   ***
  *   *           *
 *     *         *
 *     *         *
*-------*-------*----> Time
         *     *
         *     *
          *   *
           ***
ColorBurst Reference Signal
```

```
     ***
    *   *
   *     *
   *     *
--*-------*-------*--> Time
 *         *     *
 *         *     *
*           *   *
             ***
```

An Orange color 30 degrees
out of phase with ColorBurst

3. Wrapup

For now, we will end here. Look forward to a second part, which will describe the signals as measured on the 64 in more detail, and finally provide the most accurate measurements of the 64's 16 colors known. One important point which will be explained is how gamma affects RGB color measurements, and why any measurement must be given with gamma information, and why putting the same RGB values into two different monitors will not cause the same colors to be seen.

Also provided will be a program which can display the 64s colors on a PC and let you play with hue and saturation controls. When our measurements are finalized, we would expect all emulator writers to add our calibrated colors to their programs, so that the original feeling of the 64 can be retained.

Please see http://nlaredo.globalpc.net/~agonzalez/index.html
for up to date information.

Bibliography

Introduction to Digital Video, Charles Poynton.
comp.graphics FAQ.
Television Theory and Servicing, second edition. Clyde N. Herrick

```
::::::::::d:i:s:C=:o:v:e:r:y::::::::::::::::i:s:s:u:e::3:::::::::::::::::::
/S03::$d000:::::::::::::::::::S O F T W A R E:::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

: Innovation in the 90s :

Revisiting The Super Hi-Res Interlace Flexible Line Interpretation Technique

by Roland Toegel (Crossbow/Crest), Count Zero/TRC*SCS, and
George Taylor (aa601@chebucto.ns.ca)

Prelude from George Taylor, technical editor

Once you have FLI, why not just overlay some sprites on it to make better graphics? A simple idea you may have thought of, but as you will see, implementing the concept was very difficult. There are two key concepts to making it work. The following article is a brilliant piece of work on the subject. It was presented in disC=overy (issue 2) and subsequently raised a great deal of discussion. There are some confusing concepts involved so I have added extra interpretation to the original text as as editorial notes, which I hope will help bridge the gap from foreign language translated text and difficult explanations.

Prelude from Count Zero

First seen earlier this year, the new SHIFLI video technique that is described in this article is -the- paramount example of programming brilliance. The inventor of the technique, Roland Toegel (Crossbow of Crest), was helpful in providing the original SHIFLI documentation, which I translated from German into the English text that follows for the exclusive use of the disC=overy journal.

So without further ado, let us now learn from Mr. Toegel how to achieve the award winning :

'Super Hires InterLace Flexible Line Interpretation' Graphics mode !

Please note that the technique is described primarily for Commodore computers

based on the European/Australian PAL TV standard. NTSC-based Commodore machines
require an extra 2 cycles per line to be added to the raster routine.
This may require the programmer to time out the routines by hand, but this
should not be a major obstacle to overcome.

Count Zero
--

i.   Foreword

As the inventor of the SHI-FLI mode, I am pleased to have the services of
Count Zero and the disC=overy journal for the dissemination of my technique
into the English language. I must add that the text below does require the
reader to be already familiar to a high degree with VIC-II programming on the
C64. I would suggest books such as 'Mapping the 64' and the programming texts
found at ftp.funet.fi/pub/cbm/documents, for a solid base of instruction. Also,
some terms used in this document (e.g., mix-color) are meant to be uniquely
descriptive and hence, will not be found in any 'standard' programming text.
The terminology is a result of the strain that occurs when new methodology
meets old semantics. However, the experienced programmer should find the
words to be self-evident in the context which they are used.

[Mix-color: a new color derived from the original 16 by the process
of alternating normal colors at such a high rate of speed that they
blend to the eye. Also know as flashcolors. -GT]
--

1.   Introduction to Super Hires

[Hires FLI: a graphics mode which uses rasters to enable the use of a separate
color memory on each line, thus giving 2 colors chosen freely in each
8x1 hires pixel area. Interlace: alternating any two graphics modes to allow
the use of mix colors. FLI without the 'hires' modifier usually means
the use of the FLI technique with a multicolor mode as the basis. Super
Hires FLI might be more aptly named as Sprite Hires FLI, but the super means
that sprites are overlayed on top of another graphics mode. -GT]

Super-Hires Interlace FLI : The absolute successor of the Super-Hires-Modes.
Just like normal Super-Hires, the width is 96 Hires Pixels, which is equal to
12 Characters or 4 Sprites next to each other. For visual enjoyment this
area is centered though using Char-Position 15 to 26 inclusive on the
screen. The Y-Axis is 167 pixels high having nearly 21 Character lines or
8 Sprites and to have as much flexibility as possible on choosing colors or
pixels, 2 layers of sprites in a row of 4 beside each other are used
(8 sprites on the rasterline) over the bitmap graphics. Due to the fact that
all 8 sprites are used for a 96 * 21 pixel-wide area, a small multiplexer is
needed to repeat the 8 layered sprites for 8 times, at 21 pixel intervals as
you go dowards on the screen, with each row using their own bitmaps.
We thereby win 2 colors plus the 2 normal colors of the Hires-Bitmap Mode
in an 8*8 pixel block. Please note that the 2 additional colors
are the same throughout the whole picture.

2.   Super Hires with FLI !?!

FLI is, for most coders, still quite hard. Sprites over FLI, for most, quite
impossible. Maybe one or two sprites, but 8 !?! next to each other and
still FLI in each rasterline! Hard to believe, eh?

First of all you need to know how to do FLI and what it does and also
what effect sprites have on it.

2.1   Normal FLI

On each eighth rasterline (the Badlines, the first rasterline of each charline)
the VIC stops the processor for 40-43 cycles to read the new Characters and
colors of the video and color-ram. This is the case when the bits 0-2 of the
registers $D011 and $D012 are the same. Now if you change on each rasterline
the bits 4-7 of $D018, which holds the length of the video-ram (handling the
colors on bitmap graphics) and set the bits 0-2 of $D011 to get a badline on
each rasterline, you will get new colors on each rasterline in the bitmap
graphics. The only condition to be followed is that on each rasterline 23
cycles are used but for the *used cycles - 22* char of the textline the
next 3 chars have the byte $FF (light grey) read from the video ram and the
FLI effect starts after that (the FLI bug). In the multicolor mode, for the
color-ram the next byte in the program after writing to $D011 is chosen as
the color. As we are using the Hires Mode here, this is irrelevant.

> NOTE: The last 3 sentences were pretty hard to translate and I advise you
>   to read other articles about FLI aswell, if you want to know more

>   about Multicolor FLI. (CZ)

2.2   Sprites over FLI

So what does a sprite do over FLI? Pretty simple, as it just eats up some cycles. All 8 sprites use 19 cycles per rasterline, meaning in the case where we code our FLI routine without loops, we just need 2 x (LDA, STA) commands (for $D018 and $D011), using 12 cycles per rasterline. All together with the 8 used sprites that makes 31 cycles. Thus, the 'light grey' FLI-Bug occurs on char-positions 9,10 and 11 and from char 12+, the FLI effect comes up. This doesn't matter much to us, as the Super Hires Picture starts at Char-Pos 15. We therefore get 3 cycles per rasterline for other commands.

[The explanation of when the FLI effect starts in 2.1 was unclear. Here we can understand that the effect starts 3 cycles after the write to $d011. The only catch is that this code is delayed by the extra DMA time used by the sprites. Thus the formula (12+19)-22=9 gives the start of the grey pattern, and 9+3=12 is the start of the FLI effect. The constant 22 comes from a measurement of when the CPU continues after DMA is released, relative to the position on the raster line. -GT]

3.   Mulitplexing over FLI

Now we have to increase the Y-Coordinates of the sprites by 21 pixels each 21 rasterlines and give them new patterns. As we use 8 different video-rams on FLI for the colors and the sprite-pointers are always at the end of the video-ram, we are supposed to write 8 * 8 values for the patterns plus 8 values for the Y-Coordinates, resulting in 72 different addresses. Thats far too much for a single rasterline and adding the FLI routine will bust the limits. Therefore we have to do a little trick.

3.1   Changing the sprite-pointers

The trick is not to change anything at all! On the other hand we don't want the patterns to look the same everywhere. Luckily, the height of a sprite (21 pixels) is not capable of being divided by the height of a textline (8) and the smallest mutual multiple is 168 (meaning 21 * 8). As we are writing (due to the FLI) a new value to $D018 on each rasterline and we use 8 video rams, we can abuse this and have different sprite-pointers on every video-ram. The handling of where the graphics for the sprite-patterns are located becomes a little bit confusing, but it doesn't eat up any rastertime as we don't have to change the pointers.

[Put another way, we take advantage of the fact that the video matrix location is changing every raster line (and therefore also the sprite pointers at the end of the video matrix), and slice up the sprites so that the bitmap definition is located in a different place in memory for each line of the sprite. This is very confusing, and the graphics are spread throughout memory in essentially random locations to make them fit around the other graphics areas. But at least it is possible to give every part of every sprite a separate bitmap without changing sprite pointers at all, and is the first key concept to making this technique work, and a brilliant invention by Mr. Togel. -GT]

3.1.1   Table to illustrate the sprite pattern-handling

Spriteline	Video-Ram	Screenline (NOT equal to rasterline!)
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	1	9
10	2	10
11	3	11
12	4	12
13	5	13
14	6	14
15	7	15
16	8	16
17	1	17

```
18              2              18
19              3              19
20              4              20
21              5              21
--------------------------------
1               6              22

2               7              23
3               8              24

4               1              25
.               .              .
.               .              .
.               .              .
20              1              41
21              2              42
--------------------------------
1               3              43
2               4              44
3               5              45
.               .              .
.               .              .
.               .              .
```

3.1.2   Example

Small example for Sprite 0 under the following conditions:

```
Used 8 Video-Rams            : $4000 - $5FFF
content of the sprite-pointer: $43F8  80 00 00 00 00 00 00 00
                               $47F8  81 00 00 00 00 00 00 00
                               $4BF8  82 00 00 00 00 00 00 00
                               $4FF8  83 00 00 00 00 00 00 00
                               $53F8  84 00 00 00 00 00 00 00
                               $57F8  85 00 00 00 00 00 00 00
                               $5BF8  86 00 00 00 00 00 00 00
                               $5FF8  87 00 00 00 00 00 00 00
```

Thus the Sprite-Patterns are in memory from $6000-$61FF.

Therefore the pattern-handling looks like this:

Screenline	Memorylocation
1	$6000-$6002
2	$6043-$6045
3	$6086-$6088
4	$60C9-$60CB
5	$610C-$610E
6	$614F-$6151
7	$6192-$6194
8	$61D5-$61D7
9	$6018-$601A
10	$605B-$605D
11	$609E-$60A0
12	$60E1-$60E3
13	$6124-$6126
14	$6167-$6169
15	$61AA-$61AC
16	$61ED-$61EF
17	$6030-$6032
18	$6073-$6075
19	$60B6-$60B8
20	$60F9-$60FB
21	$613C-$613E
22	$6180-$6182
23	$61C3-$61C5
.	.
.	.
.	.

3.1.3   Remark to the display-routine of the editor

As the changing of the video-ram on the editor-routine happens inside of the textscreen (but the spritepointers are read inside the sideborder), the change takes effect one rasterline later.

[To state this again, the changes in the raster routine are made at a time when the sprite pointers have already been read by the VIC, therefore,

even though the changes are made on the same raster line, they don't
take effect until the next raster line. -GT]

This means that whenever the colors for the bitmap of color ram 2 are read,
the sprite pointers or video ram 1 are still active. This is the reason why
the editor uses only 167 screenlines (instead of 168) and why the first
textline of the first video-ram and the first rasterline of the bitmap
stays empty.

[I would consider this a minor bug which potentially could be fixed in
a future version, by starting the raster routine one line earlier. -GT]

3.2   Changing the Sprite Y-Values

As the changing of the sprite pointers more or less happens by itself, we just
have to make sure the correct Y-Value comes into the game. These are still
8 values, but they don't have to be set in one rasterline and we got 21
rasterlines to set them. As we have just 3 cycles left on each rasterline on
the FLI routine described and that wouldn't be enough for a simple LDA : STA,
we have to change the FLI routine a little bit.

3.2.1   Load new sprite Y-Value

As the height of all sprites is the same, we just have to do a single
LDX #$VALUE. Therefore, 2 of the 3 free cycles are used and we cannot do
anything else with the last free cycle.

```
LDX #$VALUE
LDA #$08
STA $D018
LDA #$38
STA $D011
```

3.2.2   Preload the Upcoming $D011 Value

As a STA $SPRITE0Y needs 4 cycles, we cannot include it in the next rasterline,
but we can already load the next $D011 value into the Y-Register. The free
cycle stays unused.

```
LDY #$3A
LDA #$18
STA $D018
LDA #$39
STA $D011
```

3.2.3   Write new Sprite Y-Value

As we already did the loading of the $D011 value for the next line, we now
have 5 cycles left and therefore enough time for a STA $SPRITE0Y.

```
STX $SPRITE0Y
LDA #$28
STA $D018
STY $D011
```

Now plot 3.2.2 and 3.2.3 have to be repeated for the remaining 7 sprites with
changed values for $D011 and $D018. So there is now 1 rasterline for loading
the new sprite Y-value and 8 * 2 rasterlines to write the new sprite Y-Value.
We now have 17 rasterlines and the 4 remaining ones just need an additional
NOP so that all rasterlines use the same amount of cycles.

[This tricky piece of raster code is the second brilliant concept to
making the technique work. Note also that there will be some staggering
in the timing of FLI effect, due to the unused cycles, but still
within the parameters required. Also it seems that parts of the
row of 4 sprites in 2 layers would gradually be moved downward after
parts of them have already been drawn, therefore causing a repetition
of graphics, except for the fact that the sprite pointers are different
in the new position, and that the sprites are really sliced apart in
lines, so it doesn't matter. I find this a very difficult concept. -GT]

3.2.4   Remark to the Display-routine of the editor

For simplification of the routine, which generates the FLI routine, in the
remaining 4 rasterlines an LDX #$VALUE was used instead of an NOP.

4. Memory-allocation

Now video-rams, the bitmap, and the sprites have to be placed reasonably in a VIC-Bank. As the banks from $0000 - $3FFF and $8000 - $BFFF are useless for graphics due to the overlay of the Char-rom we choose the back from $4000 - $7FFF for now.

4.1 Video-rams

The 8 video-rams need $2000 Bytes. They are located from $4000 - $5FFF.

4.2 Bitmap

The bitmap needs $1F40 Bytes. It's located at $6000 - $7F3F.

4.3 Sprites

As we need 2 sprites overlayed {two layers of 4 sprites next to each other} (four times next to each other and 8 times below each other), we need 2 * 4 * 8 sprites, meaning 64 overall. We need $1000 bytes for the sprites. We check what the video-rams and the bitmaps already allocate and recognize that only $7F40 - $7FFF, enough memory for 3 sprites, is left open. How do we rectify this situation?

As the video-rams and the bitmap just need a small part for displaying the picture, the sprites can be put into the spare parts of the video-rams and the bitmap. They have to be masked by choosing the right color in the video-rams.

A textline of a bitmap covers $140 bytes. Our Super Hires cutout just needs $60 bytes though and is centered. Thus the first and the last $70 bytes of a textline of the bitmap is free. As a sprite needs $40 bytes, we can put 2 sprites in each textline of the bitmap (one to the left and one to the right). Due to the height of the picture (21 textlines), this results in space for 42 sprites. From textline 22 on (in memory from $7A40) we can use the whole textline for sprites, resulting in 5 sprites per line. Continuing this until textline 24 inclusive, we have space for 15 additional sprites. So overall we already have 57 sprites and just 7 are missing now. These we could place in the remaining free area of the bitmap ($7E00-$7FFF), but that's not very efficient as we have some space left in the video-rams.

The Textline of a video-ram contains $28 bytes. The Super Hires cutout just needs the middle $0C bytes. As the sprites we put to the left and to the right of the picture are supposed to be invisible, we need to set a background-color in the video-ram (in our case, the color light-grey $FF). So we don't have enough spare room for the sprites to the left and the right of the picture in the video-ram.

If we finish the FLI Routine from textline 22 on and keep the video-ram on until the end of the screen (filling the this area ($4370-$43E8) with the backgroundcolor $FF to hide the sprites) we can use the remaining 7 video-rams from textline 22 (from $4770, $4B70, $4F70, $5370, $5770, $5B70, $5F70) for one sprite each. Now we have placed all 64 sprites and the allocation of the sprite pointers looks like this:

```
$43F8  80 84 85 89 8A 8E 8F 93
$47F8  94 98 99 9D 9E A2 A3 A7
$4BF8  A8 AC AD B1 B2 B6 B7 BB
$4FF8  BC C0 C1 C5 C6 CA CB CF
$53F8  D0 D4 D5 D9 DA DE DF E3
$57F8  E4 E8 E9 EA EB EC ED EE
$5BF8  EF F0 F1 F2 F3 F4 F5 F6
$5FF8  F7 1E 2E 3E 4E 5E 6E 7E
```

The pointers from $80 to $E4 are the 2 sprites which are left and right next to the picture in the bitmap.

The pointers from $E8 to $F7 are the sprites from textline 22 to 24 below the picture in the bitmap.

The pointers from $1E to $7E are the sprites from textline 22 to 24 below the picture in the video-rams.

5. Interlace

Until now we had the normal Super Hires FLI mode, supplying the basics for interlace. For the interlace mode we need 2 pictures of this kind switching, displayed 25 times per second (PAL). [30 times/sec in NTSC -GT] As such a picture fits into the VIC-Bank from $4000 - $7FFF and we have another VIC-Bank ($C000-$FFFF) with the same assumptions we can easily place the 2nd picture there.

We had in the Super Hires FLI mode (on a 8 * 1 pixel-area) the choice between 4 colors (2 sprite-colors, being the same for the whole picture + 2 FLI colors). Now, in the interlace mode, we have the choice between 16 mix-colors, meaning the combined 4 colors from picture one and two. When using interlace, mix-colors are created except for the case when the same colors are used for both pictures on the same 8 * 1 pixel-area (Check the following example) :

	Pic2 ->	Sprite1:$E	Sprite2:$0	FLI1:$6	FLI2:$9
Pic1	Sprite1:$1	$1E	$10	$16	$19
[	Sprite2:$3	$3E	$30	$36	$39
V	FLI1 :$E	$EE	$E0	$E6	$E9
	FLI2 :$6	$6E	$60	$66	$69

This results in the following 16 mixcolors:

1. White-Lightblue
2. Cyan-Lightblue
3. Lightblue-Lightblue (pure Lightblue)
4. Blue-Lightblue
5. White-Black
6. Cyan-Black
7. Lightblue-Black
8. Blue-Black
9. White-Blue
10. Cyan-Blue
11. Lightblue-Blue
12. Blue-Blue (pure Blue)
13. White-Brown
14. Cyan-Brown
15. Lightblue-Brown
16. Blue-Brown

When choosing the colors you should take care that the brightness-values of the 2 mix-colors are about the same and that they do not differ by more than 2 brightness steps, as things otherwise start to flicker too much. (e.g. Black-White flickers a lot).

Here is a table with brightness-values from light to dark.
(Colors on the same line have the same brightness)

$1	: White
$7, $D	: Yellow, Lightgreen
$3, $F	: Cyan, Lightgrey
$5, $A	: Green, Lightred
$C, $E	: Grey, Lightblue
$4, $8	: Lilac (Purple), Orange
$2, $B	: Red , Darkgrey
$6, $9	: Blue, Brown
$0	: Black

[On very old 64's (64 cycle raster line NTSC VICs), there are only 5 luminance values in the 16 colors, making them less distinct. Yellow and cyan are the same brightness, then green, grey, and purple, then red and blue. White and black form the last 2 values of brightness. The pairing of colors above are included. -GT]

6. Additional Graphics (Not handled by the editor)

We found out that on the left or right of the picture in the bitmap, $70 bytes was left for spritedata. We used just $40 bytes of that space, meaning we still have $30 bytes (6 Chars or 48 Pixels) left to both sides of the picture.

6.1 Left to the picture

Our Super Hires Picture starts at position 15. The spare $30 bytes are from position 9 to 14. As we use 14 cycles in our FLI routine and the 8 sprites use 19 cycles per rasterline, the light-grey FLI Bug now uses the chars 11, 12, 14. Thus meaning we could use char 14 for Hires FLI. On chars 9 and 10 we could just use 2 different colors (respectively 4 mix-colors for interlace) on the height of 21 textlines in the bitmap, as the FLI effect starts from Char 14 and before that no new data (colors in this case) are read from the video-ram. The colors are in the first video-ram in memory from $4008+$4009 respectively $C008+$c009.

[The shifli pic could probably be expanded by rearranging the raster code very carefully, and even areas outside the shifli effect could still use normal graphics modes, but I expect only a small improvement. -GT]

6.2   Right to the picture

Our Super Hires Picture lasts until char-position 26. The spare $30 bytes in the bitmap are from position 27 - 32. Here we could use all 6 chars for Hires FLI (or Interlace Hires FLI).

7.   Memory-allocation of a picture startable with RUN

The included SHIFLI picture, once unpacked, can be easily modified for your own use, as follows :

```
$0801-$080C Basic Startline
$080D-$0860 Routine for copying the Graphic-data to the correct memory area
$0861-$095B Routine which is setting the I/O registers and creates the
            display-routine (from $085F-$10FB)
$095C-$475B Data of the 1. Picture (to be copied to $4000)
$475C-$855B Data of the 2. Picture (to be copied to $C000)
```

8.   Conclusion

For questions or comments concerning this article :
```
Roland Toegel is available at       : toegelrd@trick.informatik.uni-stuttgart.de
Count Zero/TRC*SCS is available at : count0@mail.netwave.de
```

```
::::::::::d:i:s:C=:o:v:e:r:y::::::::::::::::i:s:s:u:e::3::::::::::::::::::::
/S04::$dd00:::::::::::::::::::S O F T W A R E:::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Defeating Digital Corrosion (aka "cracking")

- a beginners guide to software archiving

By Jan Lund Thomsen (aka QED/Triangle)

   Pontus Berg briefly touched the subject of tape cracking in his disC=overy 1 article (/S08: Software analysis and reconstructive therapy, a historical view on "cracking"). But then again, "great minds think alike". :)
   The purpose of this article is to provide a broader, more detailed view at the general components - as well as some insight on the way a cracker did his "thing" back in the old days. Although some degree of technical knowledge is required to make the most of this article the casual C64 enthusiast should also be able benefit from it.

Disclaimer

   Despite the issues discussed in this article it is in no way written to encourage software piracy. You are not allowed to use the knowledge found here to make copies for other people. If you go ahead and do this despite what I have just said it is not my problem and I can't be held responsible in any way.

[Ed. note : As always, disC=overy and the editors and staff of disC=overy can not be held responsible for the use or misuse of any information presented in this article.]

Preface

   In a world of digital corrosion the need to create backups is something most of us deal with sooner or later. You might never have had any problems yourself - but tapes or disks have been known to "go bad". And what if your tape deck suddenly blows up, rendering you unable to play your favourite games? Wouldn't it be nice to have a backup? Some people (myself included) prefer using the backup copy and storing the original in a safe place. Copies can be stored on floppy disks or hard drives for easier/faster loading. Creating an exact copy of a backup is not only very easy; it eliminates the risk of losing data due to corroded tapes.

Yes, the techniques described here are exactly the same as software pirates have been using since the dawn of time; defeating a protection scheme in order to create something that can easily be backed up. This article is *not* about Software Piracy (and I will not answer any comments on that issue). I strongly believe that owners of original software have every right to create backups.

During the course of this article I will be looking into the specifics of transferring original tape software to disk. I have several reasons for choosing the subject of tape rather than disk software:

- I have far more tape originals than disk originals. (I.e. more research material.)

- From a beginners point of view, Tape software is far easier to crack.

- Disk originals are easier to reproduce, and therefore not as vulnerable to digital corrosion as tapes.

Requirements

Apart from a tape deck and one or more original tape programs it goes without saying that a good deal of 6510 Assembly skills are required to be able to be a successful cracker. A good cartridge is really a must as well. I prefer the Action Replay Mk V or the Expert V4.2 due to their awesome debugging features.

Why don't I just use the backup option in my cartridge?

To be able to answer this question we need to go back in time and take a look at the C64 crackers of yesteryear. As different people all over the world were pirating software, cracking soon became a quest for quality rather than quantity. Of course some people cared less about quality and more about releasing a steady stream of software.
A real cracker didn't care about speed. Any cracker could produce a working copy in a couple of hours - but the real cracker wasn't satisfied with just producing a "working copy". Why would anyone capable of doing a *great* crack settle for less? For some crackers "good" simply wasn't good enough. Bugs were fixed, cheat modes installed, and excess data was surgically removed to produce shorter, cleaner cracks.
Real cracking is about technical achievement, pride, dignity, and a dedication to quality. Needless to say, no cracker worth his salt will want to freeze/backup anything. I, for one, have never used this feature of my Action Replay - not even for the purpose of producing copies for my eyes only.

Although a backup created with the "backup" option of the Action Replay, Expert, or any other freezer cartridge might run just fine, it can *never* be as good as the result produced by a dedicated cracker who knows what he is doing.

In short: *Anyone* can press a button.

Getting started

As this article is aimed at the beginner level we will only deal with "single-loaders" - i.e. programs loaded into memory in a single pass. Programs using additional I/O (highscore savers, intermission screens, levels, etc.) are known as "multi-loaders" and will be briefly mentioned in chapter 5: What's next?

Cracking (let's not beat about the bush and call it something else just to be politically correct) a tape game boils down to the following four steps:

1) Transferring the loader to disk.
2) Analysing the loader.
3) Modifying the loader and pulling the files off tape.
4) Wrapping up.

Chapter 1: Transferring the loader to disk (or "One small step...")

By definition, copy protection prevents the user from duplicating a piece of software. In the case of tape software programs are stored in a non-standard format, incorporating a fast loader of some sort. The "loader" - a short assembly program at the start of the tape contains the program code necessary to load the rest of the data. As the loader is stored in the standard tape format it is fitted with an autostart feature to prevent tampering.

Therefore, the first step to transferring a tape original to a copyable form is to load the "Loader" program *without* starting it. Luckily Commodore themselves come to our rescue with the following ROM routines:

Decimal	Hex.	Description
#63278	$F72C	Read program header off tape
#62828	$F56C	Read rest of program off tape

By inserting a tape and issuing the 'SYS 63278' command the program header will be read into the tape buffer at $033C-03FC. The first bytes of a typical program header will look something like this:

```
   .:033C 03 A7 02 04 03 4C 4F 41 .....LOA
   .:0344 44 45 52 00 00 00 00 00 DER.....
```

As you might have noticed, the filename is stored from $0341 onwards. However, the five bytes from $033C-0340 are of much more interest to us. The first byte describes the type of header: in this case "03" - a machine language program. The next four bytes contain the start and end address (Low/High byte) of the program. This header above denotes "LOADER", a machine language program loaded from $02A7-0304. As the $0300-0304 area contains a number of system vectors that can be modified to to point at the loader program and this explains the autostart.

If we modify the start/end address before reading the rest of the program (using the routine at $F56C) the file will be relocated, thereby circumventing the autostart. I recommend adding #$10 to each of the high bytes (thus shifting the entire loader $1000 bytes upwards in memory) as this makes further studies a lot easier. The file in question would be relocated to $12A7-$1304.

Having read the "loader" program to another memory location we are now able to save it to disk. But that's not all! Program code can also be stored in the tape buffer so be sure to examine the contents of $033C-03FC before moving on to step 2. If the buffer contains any other data than the five control bytes and the filename discussed earlier, save it to disk as well.

Congratulations! You have now taken the first step... there's much more to come!

Chapter 2: Analysing the loader.

Brace yourselves! This is the tough part.

There are a *lot* of different protections out there. It would be rather impossible to discuss even a fraction of them here. Some tape protections are quite easy to crack once you've gotten past the initial autostart loader. Others might require you to decrypt data in one way or another. Some particular nasty systems even load a new copy of the loader on top of the old one to prevent tampering (the Firebird Gold loader being a good example of this). Cyberload, the mother of all tape protection schemes, not only uses this technique - it also features encryption as well as two different load systems. If you can force a Cyberloader to it's knees, you have every reason to be proud of yourself.

It really goes without saying that a good deal of assembly language skills are required in order to understand what goes on. However, as long as you are able to grasp the overall function of the various parts of the program under scrutiny - you do not need to know every detail about the inner workings of it. Having said that it should be obvious that the odds increase the more you are able to understand. In other words: Practice! Practice! Practice!

Consider the following example:

```
        l1     JSR $XXXX        ; get byte from tape
               STA ($F9),Y      ; store in memory
               INC $F9          ; increase pointers
               BNE l2
               INC $FA
        l2
               [...]            ; check for end-of-file/last file/etc.
               BNE l1
               SEI
               LDA #$35
               STA $01
               JMP $1000        ; start the program.

        XXXX   [...]
        l3      LDA $DC0D
               AND #$10
               BEQ l3
               LDA $DD0D
               STX $DD07
               LSR
               LSR
               LDA #$19
               STA $DD0F
               [...]
               RTS
```

The subroutine at XXXX seems to be doing something with certain I/O registers. Understanding exactly what goes on isn't all that important because by looking at the code following the JSR (at l1) we can clearly see that something is stored in memory after calling the routine - i.e. the routine at XXXX must be some kind of loader. As Sir Arthur Conan Doyle's famous detective would put it, "Elementary, my dear Watson."

Sometimes you will have to overcome various obstacles to get to the heart of a loader. Some protections schemes rely on encryption, others on obfuscated coding. The creators of the most sophisticated protection schemes knew that understanding what goes on is the key issue regarding cracking, and went to great lengths to discourage the "professional" as well as the casual cracker.

Chapter 3: Modifying the loader and pulling the files off tape.

Now that you have penetrated the outer layers, you will want to pull the individual files off the tape and store them on disk. One approach would be to locate the piece of code that launches the main program and replace it with something that halts the computer using a infinite loop, allowing you to enter your cartridges in-built monitor, and save a memory dump to disk. In some cases it is advisable to halt the computer after each file has loaded and save the data to disk before carrying on to the next file.
Whatever approach you choose the key issue is to halt the C64 and save the relevant data.

If space permits, I prefer to insert the following piece of code:

```
        SEI
        LDA #$37
        STA $01
     l1 INC $D020
        CLC
        BCC l1
```

If there is no room in loader for big modifications like the above make a note of the original code before modifying the loader.

```
     l1 CLC
        BCC l1
```

also works quite well, not to mention only taking up three bytes. However, this requires you not to enter the monitor until you are completely sure the program has entered the infinite loop. I have used this approach so much that I know the assembly opcodes ($18,$90,$FD) by heart, even though I haven't used them in years.

Now that the C64 is halted and we are ready to save the data to disk we need to obtain the relevant addresses. I mean, why take up too much diskspace? :) Using the example from the previous chapter the data was stored at the memory location held in $F9/$FA <STA ($F9),Y>. Thus, looking at the content of $F9/$FA will give us the end address of the file. For the start-address we can either make an educated guess using the 'M'emorydump or 'I'nterrogate to scroll backwards until we encounter something that does not look like it belongs to the file (after lots of practice real crackers can be quite good at this.)
A more precise (not to mention subtle) approach is to patch the loader to store the content of the pointer elsewhere and modify the loader back to normal to prevent the start-address from being increased due to the pointer being increased.

Chapter 4: Wrapping up.

Having transfered the files to disk it is time to wrap everything up. This is by far the easiest step. Link the relevant files, then use a Char/RLE packing program (EBC, Link & Crunch, X-Terminator, etc.) followed by a "cruncher" program (DarkSqueezer, Byteboiler, Cruelcrunch, etc.)

For shorter, cleaner cracks be sure to removing any excess data. Try having a look at the game code - be sure to examine any memory-ranges that are cleared by the startup code. Often quite a bit of data can be surgically removed without causing any damage to the "patient".

Loading pictures are most often turned into stand-alone files. Just add a short piece of code to get the graphics on screen and pack everything.

Test the finished product. In the old days a lot of bad quality could have been avoided if crackers had taken the time to make sure everything worked before releasing it onto the public. Again, this boils down to real cracking being more than just churning out a heap of "warez" every day. I suppose the modern day term is "quality assurance". :)

Chapter 5: What's next?

Practice

Just because you suddenly find yourself able to breach one protection scheme doesn't mean that you have suddenly become an expert cracker. Lots of practice and further studies are needed to become successful. If you have more than one original tape using the same copy protection, don't be afraid of experimenting. Maybe you'll want to try out some new approaches. A lot can be learned from writing a program to automatically transfer files saved by a specific system. Gradually progress to other systems. If a system seems impossible to breach, try sharpening your skills on easier systems and coming back to it later.

As mentioned earlier, there are a *lot* of different loaders on the market. Although the schemes used vary from one program to another, chapters 2 and 3 should give you a rough idea about the basic idea and the steps needed to circumvent various systems.

Multiloaders

Multi-loaders (Programs using additional I/O such as intermission screens, levels, etc.) will require you to replace the original loader with a custom disk-loader. If you poke around the original loader you will most likely discover some sort of internal level-counter that can be used to refer to the name of the file you want to load next. Be sure to compress data files with a Levelpacker for a result consuming a lot less disk space. Levelpackers include a short loader/depacker program that will decompress the files on the fly.

Disk originals

Feeling confident about tape cracking? Why not move on to disks? Be prepared to encounter everything from minor obstacles to the meanest mothers in the business. Disk cracking is, as they say, "A whole different ballgame!"

The author has been a C64 enthusiast for the past 11 years - and a general 8-bit addict for even longer. In his earlier years he was part of the European C64 cracking scene. He is an active participant both on the #c-64 IRC channel, and in the comp.emulators.cbm and comp.sys.cbm newsgroups. Recently he took his group "Triangle 3532" to the World Wide Web. When he isn't busy playing Lode Runner on his C64 he is available for questions and/or general comments at the following Internet address: kwed@pip.dknet.dk

```
::::::::::d:i:s:C=:o:v:e:r:y::::::::::::::::i:s:s:u:e::3::::::::::::::::::::
/S05::$df00::::::::::::::::::::S O F T W A R E::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

A possibility to be explored : Real Time Video with the Ram Expansion Unit

by Nate Dannenberg (Natedac@dfw.net)

A few weeks ago, there was a discussion on the comp.sys.cbm usenet newsgroup concerning the viability of using the Ram Expansion Unit as a medium for displaying real-time video animation on the C64 or C128. I thought about the possibility and came to a few conclusions. The following text is how I would undertake such a project. Along the way, I shall illustrate some previous "reu-animation" efforts and how the peculiar qualities of the REU

allowed such efforts to be accomplished.
--

While I've never actually done video, Commodore did it at least twice.

On the 1750 and 1764 Test/Demo disks there is a "Pound" demo and a "Globe" demo. These two are full motion displays using your REU. The "Pound" demo is a 3-dimensional # sign transforming in and out, the "Globe" is a full rotating Earth/globe.

The controlling program for these is written in BASIC, with a small ML program to decompress the graphic data that's stored on the disk, as it's loaded into the REU.

The REU has a transfer rate of about 1,022,000 bytes per second. Since a single video frame is 1/60 of a second, that gives you 16,666 clock cycles per video frame (including video sync time).

Chances are good you will probably want to only display about 15 frames a second, which is the MPC-1 specification for PC full motion video playing from a CD-ROM disc.

With simple code, you can get about a 7.7Khz sample rate for the audio, by playing one sample byte every other raster line, and maintain a smooth 60 FPS video transfer rate, should you want to go this high.

All you have to worry about is making sure you start an animation frame transfer at the top of the intended video refresh, raster line 0. You do this with a simple wait loop:

```
loop BIT $D012
     BMI loop
```

As soon as bit 7 of $D012 goes low, the BMI will fail, signalling raster line 0 on a new frame.

The VIC chip steals 43 cycles from the CPU (and any REU transfer in progress) on the first of every 8 raster lines. The line where this occurs now only has about 21 cycles free instead of the usual 64, this is called a "bad" line in demospeak.

You want your audio routines to use the odd lines, to stay out of the way of the bad lines that occur every 8 lines. The video routines and any additional overhead can be used after the audio routine has done it's job on it's line.

An algorithm like this might do the job:

```
0 bad line, start audio REU xfer from last video refresh (see below)
1 Play 1 sample of audio data and run animation copier routine
2 ::::
3 Play 1 sample and run animation copier
4 ::::
5 Play 1 sample and run animation copier
6 ::::
7 Play 1 sample, set REU to grab 2 more audio bytes
8 bad line, start the audio REU xfer (should take 8-10 cycles)
9 Play 1 sample and run animation copier
10 ::::
11 Play 1 sample and run animation copier
12 ::::
13 Play 1 sample and run animation copier
14 ::::
15 Play 1 sample, set REU to grab two more audio bytes
16 bad line, start the audio REU xfer (8-10 cycles)
17 Play 1 sample of audio data
:
```

You use a 2-byte transfer buffer for the digi audio data, to cut REU overhead to a minimum, and to maximize the usage of those bad lines. Remember that packed RAW data, as stored in a file, takes 1 byte for 2 samples. So we're really transferring 4 samples here..

Basically on every badline you simply tell the REU to start the 2 byte transfer that was set up on the previous line. In the case of line 0 of the very first frame when starting the code, simply set up everything needed, including the first 2 byte audio pointer, which will be used by the REU at the end of line 0 to copy those 2 bytes..

This only takes 6 cycles to activate, plus 2 cycles for the dma copy. You have 11 cycles left, which may or may not be useful. Just use four NOPs and a BIT, to waste this time out to the next line.

Play 1 sample of audio data on every odd line, this takes only 6 cycles if the data from the REU was copied to Zero Page.

You have the remainder of the this rasterline and all of the next to run your animation copier, a total of about 120 cycles. You should be able to squeeze in about 80 or so bytes into this space (80 bytes plus about 40 cycles of REU overhead is about 120 cycles).

The REU can be programmed to remember where it left off after a transfer, making subsequent contiguous transfers painless. It takes about 40 cycles to set up and execute the first transfer (plus the actual copy time), and then only 6 cycles to repeat the transfer (plus transfer time of course), by using LDA #imm:STA $DF01 , where #imm is the value that tells the REU to start, I forget offhand, something like #$95 I think...

This means that you might only be able to copy 80 or so bytes the first time after an audio data copy (the first line following a bad line), but you can transfer another 80-byte block just by using only 6 cycles to re-execute the copy.

Use another 6 cycles to adjust the size of the transfer if you want more to be transferred this time, LDA #imm : STA $xfer_size_reg where #imm is your new size and xfer_size_reg is the REU register for the low byte of the transfer length..

In this way you use 12 cycles to start a new size of transfer, and are able to copy around 108 bytes, instead of just 80.

On every raster line that'S just before a badline, you play the last sample in the buffer, and use the rest of the line to set up the REU to transfer another 2 bytes of audio data into your audio buffer. This should only take about 40 cycles or so, plus whatever time is needed to update your audio data pointers.

To keep your timing easy, you should definitely keep this line down to less than 64 cycles. You can push it into the next (bad) line, but then you gotta make sure the REU has completed the 2 byte transfer before the VIC starts the line following the bad line, and you only have 21 cycles on that bad line in which to do it.

Use as much raster time as you can for the REU copier code, since the audio code is so simple and requires comparatively little CPU time.

The REU is predictable; you will always know precisely how many cycles a transfer will take. However, you may not always know how much time your own code will take, so you may need a simple wait loop like this at the ends of certain raster lines, to wait for the next raster line:

```
     LDA $D011
loop CMP $D011
     BEQ loop
```

This takes 11 cycles, plus 7 each time it loops.

Needless to say, you need to know a little about synchronizing all of this to the video display and the raster lines.

Location $D011 in the VIC is the low byte of the raster counter. Reading this register will tell you what raster line the VIC is on at any given moment.

Remember that an NTSC TV has 525 lines, which consist of two 262 line fields, often called "even" and "odd" fields. In a TV displaying a normal broadcast signal, One set of fields is normally positioned half a raster line below the other, resulting in true 525 line interlaced display at a 30Hz refresh rate.

The VIC chip gets around this by displaying the exact same data in both fields, and by placing both fields on the exact same position on screen, instead of moving one down half a line. The result is the "normal" 262 line screen with a 60Hz refresh rate.

Synchronizing to a particular raster line is a simple matter of:

```
     LDA #your_raster_line
loop CMP $D011
     BNE loop
```

This type of wait loop takes 9 cycles for the first run, plus 5 cycles each time it loops.

When you start a new frame update/copy, you need to synchronize your routine to start on raster line 0. You can do this by watching the high bit of the raster counter:

```
loop  BIT $D012
      BMI loop
```

This only takes 5 cycles per loop. As soon as bit 7 of $D012 goes low (0), the BMI will fail, signalling that the VIC has started an even video field. This way you know you are at the start of the TV's video frame as well as the VIC's idea of a frame. I don7t knwo if this would really matter, however. :)

Remember that line 0 is NOT a bad line, nor are any other lines outside the range 48 to 199, because there is no visible display here, only border.

On these lines outside the visible display, you have 43 cycles more on every 8th line, which are normally the bad lines on the visible display. They are all a full 64 cycles long like any other non-bad line elsewhere on the display.

Let's take a general routine that copies 80 bytes on every two raster lines (minus the bad line and the line before that), not using the REU's ability to remember where it left off. 80 bytes * 3 transfers per row is 240 bytes. 25 visible rows * 240 bytes yields exactly 6000 bytes.

Well we still have those 56 rasters off screen. That'S about 7 rows tall, and we can use the same routine as we do for the on-screen rows (except we have no bad lines to deal with)..

So 7 * 240 = 1680 bytes + 6000 from the on-screen rows = 7680 bytes.

That'S not quiet fast enough to do 60 FPS, but you can certainly get 15 out of it. :) In fact that looks to be good enough for 50 FPS. Well we can improve this!

Since the REU can remember where it left off, why not get it to transfer 80 bytes the first time on each row, and 108 bytes each of the next two times (as described way up there someplace)?

Well, let's add it up... 80 + 108 + 108 = 296 bytes per row. 25 rows times 296 bytes yields 7400 bytes on the visible screen. We only need another 600 bytes to finish our 8000 byte bitmap...

LeT's just use the next three rows and waste a few rasters.. 296 * 3 = 888, add the 7400 bytes we've done on the visible screen and we get 8288 bytes. Just slightly more than enough for the full screen.

After all this we still have another 32 rasters left, on which the audio routine is still running (on every other line). That audio routine is probably only using maybe 1/20 of the total raster time.

Even then we could still speed this up by using some of that leftover raster time, and with a little fiddling you can get 15Khz instead of 7.7Khz, but that'S kinda tricky and involves a lot more cycle counting.

Now remember, all of this audio and video data takes a LOT of ram....

You can only get 8 animation frames into a 64K bank in the REU if you go with full-size 320x200 bitmaps. A 1750 REU only has 8 banks, that'S a total of 64 animation frames.

At 15 FPS, that'S only about 4 seconds of video..

A 2MB REU on the other hand has 32 banks, which yields 256 frames. At 15 FPS, that'S only about 17 seconds of video.

One way to increase the video time is to use custom characters to simulate an 80x100 bitmap using a text screen. This would still be monochrome and would be much lower resolution, but you would now have about 8 times longer video. That 2MB REU would now be able to handle about 140 seconds, or just over two minutes.

The advantage of that method is you would only have to copy 1K of data instead of 8K, which could be done in about 5 rasterlines using the above copy-and-remember method.

And then you still have your audio track with that.. a 7.7Khz sample uses

7700 bytes for 2 seconds (packed 4 bit data) of sound, so a 4 second long
sample, will take just over 15K.   A 17 second sample would take about
65K. And to match up that 140 seconds of video with 140 seconds of audio
would take about 537K, or about 1/4 of the 2MB REU.

Obviously you would have to find a break-even point here, where you have
as much audio time as video time. From the looks of it, video takes about
4 times as much data as audio, when using 1K charctermapped frames, or
about 32 times as much when using full 8K bitmaps.

If you were playing the Audio data from the C64's memory, or if you had
one ram device to hold audio data and one to hold video, then the overhead
would be virtually nil. You could probably get away with a 44 KHz sample
rate and 60 FPS video, depending on the code and the speed of the ram
device(s) in question. Two REU's mapped at different addresses in memory
should in theory be enough to do it.

```
::::::::::d:i:s:C=:o:v:e:r:y:::::::::::::::i:s:s:u:e::3::::::::::::::::::
/S06::$f00d::::::::::::::::::S O F T W A R E:::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

A look into 'The Fridge'

by Stephen L. Judd (sjudd@nwu.edu)

[Ed. note : The following text is from The Fridge, an archive on the World
 Wide Web dedicated to the preservation of useful routines and algorithms.
 The Fridge (http://stratus.esam.nwu.edu/judd/fridge) is authored and
 maintained by our technical editor, Mr. Stephen L. Judd. All contributions
 to The Fridge should be sent to him at sjudd@nwu.edu]

Sample descriptions
```
     * ascii2bin.doc - A description of an ASCII to binary number converter.

     * ascii32.s     - A routine to convert a 32 bit number to ASCII and print
                       it to the screen.

     * rand1.s       - A simple pseudo-random number generator from dim4.
                       Not that great of a random sequence, but fine for
                       simple sequences.

     * bitchin.doc   - "A totally bitchin' circle algorithm"
```

--
ascii2bin.doc

The idea here is pretty simple. Numbers are read in left to right, and
each time a number is read in, the total is multiplied by ten and the
digit is added in:

```
        total=0
        digit=0
:loop   total= total*10
        total= total + digit
        read digit (and convert from ASCII to integer)
        if not EOF then goto :loop
```

There are two ways to multiply by ten. The first way is to use a general
multiplication method. The second is to realize that 10 = 8+2, thus
x*10 = x*8 + x*2 = x*2*(1+4), so with a few shifts, a temporary location, and
an addition, multiplying by ten can be done very quickly.

Example: read in the number 1653

```
        total=0
        digit=0

1st read: digit=1
        total= total*10 => total=0
        total= total+digit => total=1

2nd read: digit=6
        total= total*10 => total=10
        total= total+digit => total=16
```

```
3rd read: digit=5
        total= total*10 => total=160
        total= total+digit => total=165

4th read: digit=3
        total= total*10 => total=1650
        total= total+digit => total=1653

Voila!

SLJ 10/96
--

ascii32.s
*-------------------------------
*
* 32 bit -> ASCII conversion
*
* Take 2 -- Divide by 10 manually; remainder is coefficient
* of successive powers of 10
*
* Number to convert -> faq2..faq2+3 (lo..hi)
*
* SLJ 8/28/96

          ORG $1300

FAQ2      EQU $6A
TEMP      EQU $FE

CHROUT    EQU $FFD2

          LDA #$FF
          STA FAQ2
          STA FAQ2+1
          STA FAQ2+2
          STA FAQ2+3

          LDA #10
          STA FAQ2+4

          LDY #10
:LOOP     STY TEMP
          JSR DIV32
          LDY TEMP
          CLC
          ADC #48
          STA $0400,Y     ;Stick it on the screen
          DEY
          BNE :LOOP
          RTS


*
* Routine to divide a 32-bit number (in faq2..faq2+3) by
* the 8-bit number in faq2+4.  Result -> faq2..faq2+3, remainder
* in A.  Numbers all go lo..hi
*

DIV32     LDA #00
          LDY #$20
:LOOP     ASL FAQ2
          ROL FAQ2+1
          ROL FAQ2+2
          ROL FAQ2+3
          ROL
          CMP FAQ2+4
          BCC :DIV2
          SBC FAQ2+4
          INC FAQ2
:DIV2     DEY
          BNE :LOOP
          RTS
--

rand1.s
*-------------------------------
*
* GETRAND
*
```

```
* Generate a somewhat random repeating sequence.  I use
* a typical linear congruential algorithm
*       I(n+1) = (I(n)*a + c) mod m
* with m=65536, a=5, and c=13841 ($3611).  c was chosen
* to be a prime number near (1/2 - 1/6 sqrt(3))*m.
*
* Note that in general the higher bits are "more random"
* than the lower bits, so for instance in this program
* since only small integers (0..15, 0..39, etc.) are desired,
* they should be taken from the high byte RANDOM+1, which
* is returned in A.
*
GETRAND
         LDA RANDOM+1
         STA TEMP1
         LDA RANDOM
         ASL
         ROL TEMP1
         ASL
         ROL TEMP1
* ASL
* ROL TEMP1
* ASL
* ROL TEMP1
         CLC
         ADC RANDOM
         PHA
         LDA TEMP1
         ADC RANDOM+1
         STA RANDOM+1
         PLA
         ADC #$11
         STA RANDOM
         LDA RANDOM+1
         ADC #$36
         STA RANDOM+1
         RTS
--
```

bitchin.doc

A totally bitchin' circle algorithm

Last revised: 2/20/97 SLJ

With a revision! The old algorithm ignored one small detail -- the plot8 8-fold plotter, which can be a real pain! One thing to do is to make the algorithm draw a quarter of a circle instead of an eighth, and the way to do that is to attempt to reverse the procedure that drew the first eighth of the circle.

BLARG does exactly this; in the process, I also experimented with how the order of statements below makes a difference in the way the circles look. So the algorithm below is now the best version I know of, i.e. makes the most beautiful circles at all radii.

This is the entire algorithm:

```
        Y = Radius
        X = 0
        A = Radius/2
:loop
        Plot4(x,y)      ;Could just use Plot8
        X = X + 1
        A = A - X
        if A<0 then A=A+Y : Y=Y-1
        if X < Y then :loop

; Now more or less reverse the above to get the other eighth

        A = -R/2 - 1
:loop2
        Plot4(x,y)
        A = A + Y
        Y = Y - 1
        if A<0 then X=X+1:A=A-X
        if Y>=0 then :loop2
```

Neat!

The other possibility is to use the first half of this algorithm with an 8-fold symmetric plot, and be sure to plot the last point X=Y.

See C=Hacking Issue #9 for more details, but note that A=R/2 above, which
will make the circle correct for small R (it just rounds numbers).

--
These and other useful bits of information can be obtained through Mr. Judd's
web page, - The Fridge -, at http://stratus.esam.nwu.edu/judd/fridge

```
::::::::::d:i:s:C=:o:v:e:r:y:::::::::::::::i:s:s:u:e::3::::::::::::::::::
/S07::0100h::::::::::::::::::S O F T W A R E:::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

C-128 CP/M, trailblazer in a jungle of formats.

By

Mike Gordillo

The microcomputer world was once awash in an estimated 100 different major
operating systems with over 2000 proprietary disk format-types. The numbers
are much less intimidating today, but with the resurgence of older systems
amongst the public (IMHO), the need to extract information from an obscure
disk format may become very real to the hobbyist. Your C-128 and its
drive peripherals are extremely flexible when coupled to the right software.
When running CP/M, your C-128 is at its best in its role as an inter-platform
intermediary. This article will attempt to deliniate what your C-128, under
CP/M, can do for you in this regard.
--

The original C-128 CP/M BIOS directly supports six 5.25" CP/M MFM (Modified
Frequency Modulation) and two 5.25" CBM GCR (Group Coded Recording)
format-types. They are as follows:

```
      Epson QX10   (MFM) - CP/M Z80 DS/DD
      Epson Euro   (MFM) - CP/M Z80 DS/DD
      IBM-86       (MFM) - CP/M 8086 DS/DD
      Kaypro II    (MFM) - CP/M Z80 SS/DD
      Kaypro IV    (MFM) - CP/M Z80 DS/DD
      Osborne      (MFM) - CP/M Z80 SS/DD
      C-128 DS/DD  (GCR) - The "Default" C-128 CP/M format-type.
      C-64  SS/DD  (GCR) - C-64 CP/M (Z80 Cartridge) SS/DD
```

Note: The 1581 is "officially" supported in the 28 MAY 87 version of
C-128 CP/M 3.0+. The official 1581 format-type is MFM DS/DD.
(SS = Single Sided, DS = Double Sided, DD = Double Density)

MFM disks usually have the same number of sectors on each track while GCR
disks have a variable number of sectors for different ranges of tracks.
Commodore-DOS-style GCR is the "default" format-type for C-128 CP/M
because this allows "Joe User" to easily boot up CP/M from a 1571 or even
a 1541 (slow!) drive. (Fortunately, when Commodore finalized the 28 May 87
version of C-128 CP/M, they not only provided a format-type for use with the
1581 but added a FORMAT.COM that could create a 1581 3.5" C-128 CP/M bootable
system disk.)

What can you do with these format-types?

Taking advantage of what CP/M provides is as simple as inserting a diskette.
For example, let's say I have some files on my Kaypro which I need to use on
C-128 CP/M. All I have to do is put the Kaypro disk in my trusty 1571 and
voila, I can use it and the files within as I please with no hassles. This
is no small point of trivia. The message here is that you can use data on
these "alien" format-types as if the data were sitting on the "default"
C-128 CP/M disk.

What about copying them over to C-128 CP/M disks?

No problem! The transfer process is as easy as using an off-the-shelf file
copier because as far as CP/M is concerned, the "alien" and "default" (native)
format-types are one in the same. This degree of transparency is due largely
to the internal disk format table which exists within the computer's memory
as part of the CP/M BIOS (Basic Input/Output System). This arrangement
constitutes better support and flexibility over the rigid Commodore DOS system
when it comes to juggling different format-types.

What happens if I run across a format-type that is not directly supported by

C-128 CP/M?

Flexibility to the rescue! The "built-in" drive table is modifiable (both in memory or on the system disk). If you need to access other CP/M format-types, the solution may be as easy as installing the parameters that match the disks in question. A good example of a utility that does this is the UNIDRIVE utility by Frank Prindle. It reconfigures the drive table, allowing up to ten different 5.25 MFM format-types at any one time from an overall selection of twenty-four. Because of the drive table, UNIDRIVE is a simple elegant program that can make these changes without sacrificing any TPA (Transient Program Area) memory. The undisputed king of format "jugglers" and nowhere near as simple is the JUGGLER utility by Miklos Garamszeghy. This program can read and write and format over ONE HUNDRED THIRTY different format-types, mostly 5.25" MFM (with 1571) but also 3.5" MFM (with 1581) based disks. This includes the popular space extending, speedy alternative formats for C-128 CP/M, namely Maxi 1571/1581 and MG 1581. A demo version of the JUGGLER utility is publically available. It does not have as many features as the full version, you can only access twenty-two format-types and modifying the "built-in" drive table is out of the question. This would be inconvenient if the full version were hard to find as it once was. This has changed as its author as put up the full version of Juggler on the World Wide Web at http://www.herne.com/cpm.htm

What about MSDOS MFM disks, can I use them as well?

Yes you can! I know of at least three separate utilities that will allow communication between C-128 CP/M and MSDOS diskettes. First on the list is the RDMS 2.33 utility. This is an old program designed to read MSDOS diskettes, and that's all it does. It will read from MSDOS into CP/M but will not write back to MSDOS. Frank Prindle ported over this program (for the 1571 drive) and he would have probably added a "write MSDOS" option if someone else hadn't beaten him to the punchline. The TRANSFER 128 v1.2c utility (ported over by Gilly Cabral from David Koski's original TRANSFER util.) came soon after Prindle's rendition of RDMS 2.33. In fact, the routines used to communicate with the 1571 are based on Prindle's work. TRANSFER 128, however, goes way beyond RDMS 2.33. Not only does it read and write to MSDOS disks but it also provides a suprisingly high level of sophisticated access to the internal workings of MSDOS disks. For example, it can display the MSDOS FAT (File Allocation Table) and reconstruct it, if needed, using a backup FAT as a template. TRANSFER 128 supports single and double sided 5.25" format types with eight or nine sectors per track. It places no limits on the size of the data it can transfer. The only limitation is the free space left on the destination diskette. This is a huge advantage over the first "native" mode MSDOS to CBM DOS programs which use internal memory buffering. For many years, TRANSFER 128 was my only standby for large files. When better utilities came out for the "native" modes, I still found TRANSFER 128 to be my choice. Why? Because of the excellent Ram Expansion support under the CP/M BIOS. CP/M thinks the Ram Expansion Unit is truly a real drive. TRANSFER 128 (and other CP/M programs) does not have to worry about steering clear of "interface pages" and silly stuff like that when dealing with the additional Ram. Eventually, people wrote transfer utilities that did work with CBM RAMDOS, for example. Too late, my heart was set on TRANSFER 128. It's been a good companion and since it comes bundled with its Turbo Pascal source, I can always modify it in case of trouble. In fact, I had to do just that when I set up several BIOS/BDOS/CCP enhancements on my system. (But that's another story :)))
The one tiny snag with TRANSFER 128, however, is that it will format the various MSDOS format-types perfectly as far as MSDOS itself is concerned, but "native" mode transfer programs do not recognize the disks which TRANSFER 128 has formatted. I am at a loss to explain this, so I won't ;), but its safe to say i'll give TRANSFER 128 the benefit of the doubt anytime!

What about 3.5" MSDOS format-types using my 1581?

Ah, yes, I've neglected to mention what may be the best "transfer" utility available for the C-128. It may be too sophisticated for young viewers so this section of this article will be rated PG-13 :).

Well Well Well !??!? Don't keep us in suspense!

Not to worry! The utility i'm harping about is called MSDOSEM (by Nichita Sandru). It doesn't just give you a temporary transfer "window of opportunity", it makes CP/M think of MSDOS disks as native CP/M disks and it provides (get this) *full* subdirectory support!!!

This is similar to the transparent "juggling" of CP/M format-types, yes?

Yes, except that CP/M disks carry the same directory structure no matter how their format-type is arranged. In order to support the MSDOS directory structure, MSDOSEM creates a separate buffer area for internal conversions between CP/M and MSDOS logical constructs. The actual physical nature of the MSDOS format-types, however, is handled (once again) with a modified CP/M drive table. MSDOSEM has no problem with 5.25" (1571) or 3.5" (1581) double density MSDOS format-types. Again, as described earlier in this article, transparency means that you can play with the "alien" disk as if it were a "default" one. Any copy utility you have sitting around can now access MSDOS diskettes with MSDOSEM as the go-between. This is the ultimate in MSDOS to C-128 transferring because you (essentially) do not have to transfer anything anymore! The "stuff" on your MSDOS disks is no longer anathema to CP/M, even though it may be to you :).

What about support for regular Commodore DOS (GCR) disks?

In reality, the original "default" C-128 CP/M format-type is as Commodore DOSian GCRish as anything Commodore has done for the "native" modes. I take advantage of this when backing up my CP/M disks, either with FastHack'em (or any GCR copier) in C-128 mode or IMAGE.COM under CP/M itself. These programs allow us to manipulate Commodore GCR on a gross level. For a more refined manipulation of files "stuck" in regular CBM GCR disks, I generally use the RDCBM 2.1 utility by Rob Tillotson and Turbo Penguin Software. This program can read files from 1541 or 1571 "native" Commodore GCR disks and spit them out as straight binaries (eg., no translation) or it can convert files from

PETSCII to ASCII. Like RDMS 2.33, RDCBM 2.1 does not write back to the disks it so aptly read and it does not read anything on the 1581. These quirks are *highly irritating* considering its a pretty good program otherwise. Good enough to provide limited support for files under the GEOS file structure. Fortunately, I surmise that I have not run into a more recent version of RDCBM to evaluate. I say this because the source listing of this program contains "Hooks and Crannies" pointing to the author's efforts to implement a "write-back" option and 1581 support.

You mean I cannot transfer material from my CP/M disks to my regular 15xx disks???

No I don't. There exist several "native" C-64 and C-128 utilities which will do this. The C-128 "native" mode version of the Big Blue Reader program has this ability. Utilities like Supersweep and Crosslink can also port your files to CBM DOS 15xx disks, albeit these two programs set limitations on the size of the files in question.

That's nice, but i'm a C-128 CP/M'er through and through!

Ok, you beat it out of me. There *is* a way to "save" files to a regular CBM DOS diskette from within C-128 CP/M. You first use the C128LOAD utility by David Bratton to load a file into Bank 1 of the CP/M memory map. Next, you hit CTRL-ENTER (on the numeric keypad, do not press RETURN) and this will put you in C-128 "native" mode. Finally, you use the built in C-128 ML monitor to save to disk the memory block (Bank 1) where the program was loaded. Please note the following example;

```
     A> C128LOAD DIEHARD.C64

        Program loaded - from 2000 to 8192  <-- Last byte-address

     --When the A> prompt reappears, press the CONTROL and ENTER keys
       simultaneously.  (The ENTER key is on the NUMERIC KEYPAD.)

       <The C-128 will now boot Basic 7.0 -- C-128 "Native" mode>

     READY.
     MONITOR (activate the C-128 ML monitor & insert CBM DOS disk into drive 8)

     S "DIEHARD.C64",8,2000,8193  <-- Always add +1 to the last byte-address.
                            ----
```

In effect, we have just moved the file DIEHARD.C64 from C-128 CP/M to a CBM DOS disk (remember to use the bank address of 1 when saving). While this method places an absolute restriction on the file size of what you can "save", it means that files can be "saved" to any drive supported by the C-128. It is possible to save only one file per iteration of this cycle. In order to save many files, you will have to reboot CP/M and go through these steps for each file involved. This is a roundabout way to "write-back" to CBM DOS and it certainly isn't as contained within CP/M as most of us would like.

How do the exciting CMD FD-2000 and FD-4000 drives factor into all this?

They are "covered" insofar the programs that support the 1581 also support them, namely the JUGGLER, MSDOSEM, and C128LOAD utilities, and also the 28 MAY 87 Version of CP/M 3.0+ on the C-128. The FD series drives do a pretty good job at emulating the 1581 and should present no problems under C-128 CP/M. The 1581 is limited to around 800 kilobytes and this may not be adequate for you.

As far as accessing the high capacity disks within the scope of the FD-drives, the key issue is software (again). For example, there is nothing in terms of hardware that says we cannot use these drives to play with MSDOS 2.88 megabyte formats (ala FD-4000) or create 2.88+ megabyte CP/M formats of our own design. I do not know if FD-based software exists to allow this under C-128 CP/M.

How do I use the strengths of some programs to get around the limitations of other programs?

I take it you are asking this because of the software gap in linking CP/M to CBM DOS (at least from the CP/M side of the equation)? Well, we have a trump card in our sleeves in the form of the MSDOS format-types. These tend to be the lowest common denominator for information inter-change in the entire microcomputer world. For example, if the JUGGLER utility lacks the ability to communicate with a particular brand of CP/M format-type (a rare event indeed), chances are the system (with the funny format-type) has some means to get information from its disks onto MSDOS disks. Obviously, I can simply use the MSDOS format-types as a bridge "over troubled sectors" :) and not worry anymore. The same holds true for CP/M to CBM DOS. If it becomes too inconvenient to save with C128LOAD and the built in C-128 mode monitor, I simply pop up TRANSFER 128 or MSDOSEM and let them put things into MSDOS disks for later retrieval by CBM DOS-able "native" mode programs such as Big Blue Reader or Little Red Reader/Writer 128. Mind you, this has its own share of inconveniences, but it does present an attractive alternative to some people.

What if I don't have a C-128 but I need to put files on a C-128 CP/M disk or some other type of CP/M disk?

Hmmm...that's a tad outside the scope of this article but I can see a situation arising from this where a C-128 CP/M user might miss out on some software. If you have a machine that runs MSDOS, you can avoid that situation. There is a MSDOS to CP/M utility called 22DISK140 which supports 140 different CP/M MFM formats in its demo version and over 400 (!) in the full version. The key thing here is that it supports the Commodore 1581 CP/M MFM format. Any files you put in there are going to be useful to a C-128 CP/M user as long as he/she has a 1581 and the 28 MAY 87 Version of CP/M 3.0+ on the C-128 (or a suitable patch which supports the Commodore 1581 CP/M MFM format).

You've shown me the software, but where do I get it?

All the C-128 CP/M utilities I mentioned are available on Internet FTP sites such as ccnga.uwaterloo.ca in directory /pub/cbm/os/cpm. I have also seen some of them in the GEnie and Delphi information networks. The 22DISK140 utility is available on the Internet FTP site oak.oakland.edu in directory /pub/msdos/diskutil

Final thoughts...

Software Software Software... This article is by no means the final word on what you can or can't "port" out of C-128 CP/M. For example, Apple CP/M GCR disks are completely isolated from C-128 CP/M, 22DISK140 or anything out there besides an Apple drive! However, some clever magician among you may set to master the Apple GCR format using a Commodore GCR drive. If you do it, drop me a line, good CP/M software is usually a thing of beauty. As I hinted the last time we met, CP/M's strength lies in its software, (: and in its users does its software lie. :)

Mike Gordillo is an expert in CP/M and Z80 programming as well as a devout Commodore fanatic. He may be reached on the Internet as s0621126@dominic.barry.edu for general comments or questions.

```
::::::::::d:i:s:C=:o:v:e:r:y:::::::::::::::i:s:s:u:e::3::::::::::::::::::
\H01::::::::::::::::::::::::::H:A:R:D:W:A:R:E::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

The X10 PowerHouse, What is it?

by Dan Barber (xy3951@epix.net)

Well to put simply, it is a system that allows you to control up to 256 lights and or appliances in your house automatically, or from a distance.

The most basic part of the system is the modules. They are small boxes that plug into a plain 2 or 3 pin electrical outlet (depending on the module you have purchased) and the device you wish to control is plugged into the module.

Then you can get numerous remote controls that send signals to a receiver that then sends the signals over your wiring to the individual modules, which then turn on, off, or dim (if you are using a lamp module).

As I have said, you can get remote controls to control the appliances, but why not just control them with the normal switch and save yourself some money? Because a computer interface can also be purchased for the system, and this interface can be programmed with a Commodore 64/128! Or if you are one of the unfortunate that don't own a Commodore, then you can use a MS-DOS, Apple Macintosh, or Apple IIe/IIc.

The CP290 Home Control Interface is 5 inches wide by 7 long, and 1 1/2 deep. The top face is covered by rocker keys which allow you to control 8 modules from one HOUSECODE which is adjustable. The modules have two dials on them, the first is the letters A-O. This is called the the HOUSECODE. The second dial on the modules is the number 1-15, this is the UNITCODE. So, your first module would probably be name d A1 and the second A2 and so on, for a total of 256 modules.

The interface can handle control of all 256 modules but if you are using the Commodore 64/128, you can only program 95 using the included point-and -click GUI (Graphic User Interface) program.   If you have more than 95 modules you will have to use the included BASIC extension program that is included on the disk. Then you can create your own simple program to control all of the modules if need be. But I can't personally imagine anyone using over 95 modules, unless you have a mansion. :)

The program that sets up the interface is very simple to use. With the power off you connect the cable to the userport and the interface turn it on and load the program. You set the time and the HOUSECODE you want the rocker keys to respond to and then the screen shows the different rooms you can "install" modules in. There are 9 "virtual" rooms you can place and program modules, though it does not make any difference what room you have the module in as long as the dials on it are set properly. Programs for use with other computer systems have differences, some you type in the name of the room instead of using the "virtual" rooms. And some can control all 256 modules with the supplied software, where as other systems must write there own software if they need to control all 256 modules.

Once you pick a room (by using the cursor keys or the joystick to move a pointer and clicking) you are presented with 11 "tabs" where you can "place" modules. When you select one you are presented with choices of icons (graphics) that look like appliances and lamps. When you chose one it is stored in the interface as well as the HOUSECODE and UNITCODE. Note, it does not make any difference what icon you select to go along with a module, it is just to remind you of what the device is. So you could have a toaster icon (graphic) actually controlling a coffee pot, the only thing that really matters is the dials on the modules and the interface codes are the same.

Once you have the rooms set up with icons you like, you go to the operate mode. This is where you program the interface to come on or off at specific times of days. You can set it to go on NOW, which turns it on immediately. OFF immediately, ON TODAY, OFF TODAY, ON/OFF SPECIFIC DAYS, ON/OFF EVERYDAY. And this is the beauty of the system, you can have your house have a "lived-in" look even though it isn't. With SPECIFIC DAYS and EVERYDAY modes, you can set a security mode. This means that if you set something to come on at 5:00pm, it would come on between 5:00pm and 5:30pm. It just picks a random time and ON (or OFF) it goes. The interface is very versatile and you can enter numerous ON and OFF times, so for instance, if you have kids that forget to turn off the lights you can have them go off-only at designated times, the same goes for ON. Now this is why this system shines over others, when you aredone programming you unplug the interface from the computer! It does not tie the computer up. You only use the computer to program the interface. Afterwards it runs completely on its own.

The interface will handle up to 128 timed events, which means if you were controlling a Christmas light setup, and you were changing the lights every minute (the closest span that you can program a timed event) you could have over 2 hours of automatic control. I should mention that it does not make any difference how many modules have been setup to go on or off for one particular time (up to 16 modules). It would still be considered one time event (ex. modules A1, A4, A7, and A15 programmed to go on at 70% brightness on Mondays, Wednesdays, and Fridays at 7:30 p.m. is just one event). Also, you can only set brightness for a lamp module with the interface hooked up to the computer to send the commands manually, or as a pre-programmed event. So, if you just hit one of the rocker keys or turn the lamp on, off, and back on (the module will sense this and turn on), the lamp will just come on full.

AFTER PROGRAMMING IS COMPLETE

And after you get the interface all programmed just the way you want it-the power fails and you loose your entire program. Well, if you have a 9 volt battery installed the program would have been safe. The battery can hold the memory of the interface for 10 hours (or so). And it is a good precaution, the only problem might be to forget it is in there and after a few years, the battery leaks. But that applies to any battery-powered device. There is also the option to save the entire contents of the interface to disk. This is available as a separate program on the disk, and it saves the interface data as a file to the disk. Three saves can be on the disk at one time, and if you need more just copy the disk, it is not protected. One of the greatest benefits of this feature is the ability to swap setups in and out so you can have one for different times of the year. For instance you could save your Christmas light setup to disk and next year get out the same setup and save yourself a lot of time in programming.

If you have a elaborate house setup that takes 20 minutes or more to program. It is great to be able to load the settings back in a minute or so! Another good reason for having a backup is because the interface can "lock-up" and you must remove the battery and unplug it to reset it. This of course causes the program to be lost so having a backup is necessary in this case, otherwise you have to reprogram it. I have had it happen once, but I was moving the interface. I think that the battery lost contact momentarily while I was moving it to another room and caused a "lock-up" to occur. But I had just done a backup so nothing was lost.

Power failures have different effects on the modules as well. The appliance module will stay in the same position (they have a latching mechanical relay type switches) as it was before the outage. Where as, the lamp modules will be off after a power failure (they have a triac for dimming), no matter what state the module was in before the failure.

HARDWARE

There are a lot of clones on the market, and they all appear to be compatible. X-10 was the first, but other places such as Radio Shack has both the interface, modules, and other addons for the system. But in the case of Radio Shack's interface, it is missing a very important feature (at least in my opinion): no manual rocker keys! Other than that, I don't know of any differences between the systems.

** Just as this issue of disC=overy was going to press, I found out that X-10 manufactures all the clones themselves! The differences are minor and limited to cosmetic changes as the example listed above.

The equipment is long lasting (people have had the system in place for over 20 years). But of course if you have a lightning strike I would not expect them to survive, and don't try to put a surge suppressor on them either. Most filter the lines and will block any signals going out or coming in to the attached equipment. And a plug-in intercom system or baby monitor will interfere with the signals if they happen to be sent at the same time as the intercom is on.

In short this system is MUCH easier than running cabling all over your house, and much cheaper. You can get the interface and 4 modules for around $100. And if you want to control a chandler or other such lighting fixture, just replace the switch with an X-10 controllable switch, and the same applies for outlets. And replacing them is very easy, and can be done in an afternoon (unless you are doing an entire mansion). Just make sure to turn off the power at the breaker!!!! Other addons for X-10 include entire alarm systems, motion sensing flood lights, emergency dialer (dial for assistance when you push the remote control), telephone responder (remote control your home while you are away), and many other interesting products.

There have been some interesting uses I have heard of for the X-10 system. Like, magicians using it in there act, remote control lighting for underground cavetours, and lighting control for chicken farms to fool the chickens into laying more eggs. And many others, this proves that is no limit to what the system can do if you put your mind to it.

TECHNICAL SEPCIFICATIONS

The processor is in the interface is a 80C48. And I mentioned before a maximum of 256 modules with a maximum of 128 timed events.

The transmissions are complex and are sent twice for verification and basically the signals involve short RF burst which represent digital information. The transmissions are synchronized to the zero crossing point of the AC power line. The goal is to transmit as close to zero crossing point as possible, but certainly within 200 microseconds of the crossing point. A Binary 1 is represented by a 1 millisecond burst of 120 kHz at the zero crossing point, and Binary 0 by the absence of 120 khz. And the instructions to the module is sent with those two signals (as is all computer information).

The complete code transmission involves eleven cycles of the power line. The different cycles represent different commands to the module. Such as the first two cycles represent start, next four represent the House Code and the last five represent either a Number Code (1 thru 16) or a Function code (ON, OFF, etc). The programming manual is reproduced below with permission and for the edification of the reader.
--

[Ed. Note : Portions of the technical information below require some basic understanding of signal theory and electronics.]

INTRODUCTION
Software is required to use your computer to program the X-10 Home control Interface. The interface is packaged with software and connecting cable for either IBM PC, Apple Macintosh, Apple IIe/IIc or Commodore 64/128.
THAT'S ALL YOU NEED.

A utility program of Basic statements is included with the Home Control Software for IBM, Apple IIe/IIc and Commodore Computers. These Utility programs let you write your own programs in Basic.

For more advanced programming you may also need to refer to this programing guide.

This programing Guide is for advanced programmers who wish to write their own software using Machine code and need information than is supplied in the owner's manual which comes with the X-10 Home Control Software.

If you do NOT intend to write your own software, don't be intimidated by this programming guide- you don't need it.

The Interface must be connected to your computer for programming but once programmed it can be disconnected and will continue to send commands to the X-10 Modules under time control.

The Main functions of the Interface are summarized as follows.

*Maintains a real time clock
*Stores the timed events relating to control of lights and appliances in the home.
*Stores the graphics data required by the computer to display the details of lamps and appliances installed into the house by the user.
*Transmits X-10 Control Signals onto existing house wiring to control lights and appliances connected to the X-10 Modules.

A 9volt Alkaline battery will provide approximately 100 hours back up for the Interface clock and stored data. When the Interface is running on battery power, the L.E.D. pulses approximately once every 5 seconds.

Up to 128 timed events + 256 ICONS (Graphical pictures of lights and appliances) can be stored in the Interface. A timer event is any number of unit codes on the same Housecode programmed to go on or off at a particular time at a specified brightness level on any day or days of the week.
(E.G. Modules A1, A4, A7 and A15 programmed to go on at 70% brightness on Mondays,Wednesdays and Fridays at 7:30 P.M. is just one event and 128 events can be stored.)

The Interface has 8 rocker keys to give manual control of unit codes 1 thru 8 onthe Base Housecode. Base Housecode is set to "A" on power up but can be changed by the software.

PROGRAMMING

The Interface is programmed to recognize 8 different types of instruction for the computer and each instruction has an ID number between 0 and 7. Each instruction from the computer has a leading SYNC pattern of 16 x FF bytes. The ID number tells the Interface what type of data to expect and a check sum is maintained which is compared with the last byte of data in the instruction. If the check sums agree, the Interface will acknowledge back to the computer and obey the instruction. It is suggested that if no response to an instruction is received within 10 seconds, the computer should advise the user of potential problem with the connections to the Interface.
(E. G. the program could display a message such as "Error check Interface connections. Press Enter to continue".)

The Interface is programmed via the 5 pin DIN socket on the back of the Interface. The pin connections are shown below.

```
                        Looking at Back of Interface
     5          1
       4     2
          3
Pin  Description
1    -
2    Receive (Input)
3    Ground
4    Transmit (Output)
```

The input signals from the computer (receive data input) are connected between pins 3 and 2.

The output signals to the computer (transmit data output) are connected between pins 3 and 4.

A data cable is available for IBM, Macintosh, Apple IIe, and Commodore 64/128 and is included with the Interface and software for these computers. <Note-The Commodore cable and software must be purchased separately, it is no longer sold with the interface. To get the cable-software package you must order one from X-10..>

Voltage levels meet RS-232 specifications and the data format is RS-232 with the following characteristics.

```
Baud rate:  600.
Data bits:   8.
Parity:      None.
Stop bits:   1.
```

BYTE FORMAT

```
           Start                                Stop
            Bit  D0  D1  D2  D3  D4  D5  D6  D7   Bit
     V+    ---     ---            ---     ---
           ! 0 ! 1  ! 0   1   1   1 ! 0 ! 1 ! 0 !  1
     V- ---     ---    -----------     ---    ---

1011   D   D0 to D3
1010   5   D4 to D7

Dec = 93
Hex = 5D
0 = Mark
1 = Space
```

A gap of 1 Millisecond should be left between each byte of data sent.

A start bit signifies that a string of 8 data bits will follow. A start bit is always a SPACE bit, i.e. "0". A stop bit signifies that the data is finished and separates one byte from another. A stop bit is always a MARK bit, "1".

DOWNLOAD BASE HOUSECODE

When the Interface is first powered up, the Base Housecode is set to "A". To change this you must first send a leading SYNC pattern of 16 x FF bytes to the interface, followed by the identifier "0" for "download base Housecode" and then of data, the upper nibble of which contains the Housecode information. See below.

```
BYTE    D7    D6    D5    D4    D3    D2    D1    D0
1-16    1      1     1     1     1     1     1     1  Sync, 16 X FF
 17     0      0     0     0     0     0     0     0  ID 0, download Base
                                                      Housecode
 18    ---HOUSECODE------------- 0     0     0     0  See table 1.
```

TABLE 1

House	Byte 18	House	Byte 18	House	Byte 18
A	60	B	E0	C	20
D	A0	E	10	F	90
G	50	H	D0	I	70
J	F0	K	30	L	B0
M	00	N	80	O	40
P	C0				

Base Housecode is used by the rocker keys on the Interface. Changing the Base Housecode will reset all timer events and graphics data stored in the Interface, therefore before downloading a new Base Housecode to the Interface, the program should warn the user of this. (E.G. the program could display "Warning changing Base Housecode will erase all program information. Continue with change yes/no".)

After a successful download, the Interface will acknowledge by sending the "ACK" message to the computer.

ACK MESSAGE

Byte	D7	D6	D5	D3	D2	D1	D0	
1-6	1	1	1	1	1	1	1	FF X 6
7	0	0	0	0	0	0	S	Status

The STATUS bit is reset to "0" during power up of the Interface and is set to "1" by a download of data from the computer (any data with byte 17 equal to ID 0, 1, 2, or 3). The STATUS bit is used to warn the computer that the Interface has been powered down. E.G. a STATUS bit equal to "0" could tell the program to display a message such as "The Interface has been powered down and contains no data. Press Enter to continue".

DIRECT COMMAND (instant ON or OFF)

To turn something ON or OFF or adjust the brightness level of a light instantly,m it is first necessary to send a leading SYNC pattern of 16 x FF bytes of data to the interface. This is then followed by the identifier "1" for "direct command" and then 4 bytes of data followed by a check sum. The check sum is the sum of bytes 18 through 21. See below.

BYTE	D7	D6	D5	D4	D3	D2	D1	D0	
1-16	1	1	1	1	1	1	1	1	SYNC FF x 16.
17	0	0	0	0	0	0	0	0	ID1, Direct command.
18	LEVEL				FUNCTION				See notes 1 and 2.
19	HOUSECODE				0	0	0	0	See table 1.
20	9	10	11	12	13	14	15	16	Bit mapped unit codes.
21	1	2	3	4	5	6	7	8	X-10 Modules.
22	CHECK SUM								Sum of bytes 18-21.

NOTE 1
"LEVEL" is dimmer settings for lamps. This applies only to X-10 Lamp Modules and Wall Switch Modules. Level F Hex is full DIM and level 0 Hex if full BRIGHT. The lamps specified by bytes 20 and 21 will switch on, adjust to full brightness and then DIM to level specified by the upper nibble of byte 18. All codes between 0 Hex and F Hex are acceptable thus providing 16 discrete light levels.

NOTE 2

D3	D2	D1	D0	Function	Explanation
0	0	1	0	ON	Modules with house codes as specified by upper nibble byte 19 and unit codes as specified by bytes 20 and 21 will turn on.
0	0	1	1	OFF	As above, except turn OFF.
0	1	0	1	DIM	Lamp Modules and Wall Switch Modules addressed as above will turn on, adjust to full intensity and then DIM to the level specified by upper nibble of byte 18. Appliance Modules do not respond to bright and dim codes.

NOTE 3
If the check sum is accepted, the Interface will send the ACK response to the computer and will then transmit the X-10 codes onto the house wiring. When the power line transmission is complete the command is uploaded to the computer (see command upload). Also, if X-10 codes are transmitted by pressing the keys on the Interface, at the end of each transmission the codes are uploaded to the computer. This allows the computer to keep track of the ON/OFF status of the Modules while it is connected to the Interface.

DIRECT COMMAND EXAMPLES
Example 1: Turn ON modules A1 and A4

Bytes:	1-16	17	18	19	20	21	22
Data:	FF	01	02	60	00	90	F2

Example 2: Turn OFF modules A1 and A4

Data	FF	01	03	60	00	90	F3

Example 3: Turn on lamp module B9 and DIM to 50%

Data	FF	01	75	E0	80	00	D5

Example 4: Turn OFF all modules with housecode A

Data	FF	01	03	60	FF	FF	61

COMMAND UPLOAD (Interface to Computer)
This follows every transmission of X-10 onto the power line either from pressing the rocker keys, or from direct commands, or from timed events. This enables the computer to keep track of the ON/OFF status of lights and appliances.

Byte	D7	D6	D5	D4	D3	D2	D1	D0	
1-6	1	1	1	1	1	1	1	1	FF X 6.
7	0	0	0	0	0	0	0	0	Status.
8	HOUSECODE				FUNCTION				See note 4 below.
9	9	10	11	12	13	14	15	16	Bit mapped unit codes of
10	1	2	3	4	5	6	7	8	X-10 Modules.
11	BASE HOUSECODE				0	0	0	0	Same as table 1.
12	CHECK SUM								Sum of bytes 8-11.

NOTE 4 -House code same as table 1. Function same as note 2, except that the code for DIM is UPLOADED to the computer as 0100 (4hex).

SET CLOCK (Computer To Interface)
To set the clock in the Interface it is first necessary to send a leading SYNC pattern of 16 X FF bytes of data. This is followed by the identifier "2" for set clock and then 3 bytes of data followed by a check sum. This check sum is the sum of bytes 18 thru 20. See below.

Byte	D7	D6	D5	D4	D3	D2	D1	D0	
1-16	1	1	1	1	1	1	1	1	SYNC 16 X FF.
17	0	0	0	0	0	0	1	0	ID2, set for clock.
18	0	0	MINUTES						HEX 00 to 3B (0 TO 59).
19	0	0	0	HOURS					HEX 00 to 17 (0 TO 23).
20	0	Sun	Sat	Fri	Thu	Wed	Tue	Mon	Bit mapped Days.
21	CHECK SUM								Sum of bytes 18 to 20.

SET CLOCK EXAMPLES

EXAMPLE 1 To set clock to 9:30 a.m. on Monday.

BYTE	1-16	17	18	19	20	21
DATA	FF	02	1E	09	01	28

EXAMPLE 2 To set clock to 7:45 p.m. on Friday.

BYTE	1-16	17	18	19	20	21
DATA	FF	02	2D	13	10	50

TIMER EVENT OR GRAPHICS DATA DOWNLOAD
(Computer to Interface)

To download either a timed event or graphics data you must frist send a leading sync pattern of 16 X FF bytes. The ID. (Byte 17) is 3 HEX for both timer events and graphics data but D2 in Byte 19 is a "0" for timer events and a "1" for graphics data.

Timer events are stored in bytes 0 to 1023 of the 2k X 8 RAM in the interface. Only bytes 20 to 27 of the downloadable message are stored. Each event (group of 8 bytes) is assigned a Start Address in the RAM in the Interface. This Star Address is specified by A0-A4 in Byte 18 and A5-A6 in Byte 19. D0, D1, and D2 in byte 18 must ALWAYS be 0, so that the Start Addresses increase in multiples of 8 (0, 8, 16, ... 1016). The computer should keep track of the event Start Addresses and load new events into vacant address locations in RAM. Byte 20 designates the type of timer event as shown in table 4. Bytes 21 through 23 set the time and day of the event. Bytes 24 and 25 specify which Modules will be controlled and byte 26 specifies the Housecode of these Modules. Byte 27 specifies whether the Module(s) will turn ON, OFF, or DIM and to what brightness level. Byte 28 is the sum of bytes 20 to 27.

TIMER EVENT DOWNLOAD

BYTE	D7	D6	D5	D4	D3	D2	D1	D0	
1-16	1	1	1	1	1	1	1	1	SYNC 16 X FF

17	0	0	0	0	0	0	1	1	ID3, event/graphics download.
18	A4	A3	A2	A1	A0	0	0	0	A0 to A6 binary coding of
19	x	x	x	x	x	0	A6	A5	event number.
20	0	0	0	0	MODE				See table 4.
21	0	Sun	Sat	Fri	Thu	Wed	Tue	Mon	Bit map of days.
22	0	0	HOUR						HEX 00 to 17 (0 to 23).
23	0	0	0	MINUTE					HEX 00 to 3B (0 to 59).
24	1	2	3	4	5	6	7	8	Bit map of unit codes.
25	9	10	11	12	13	14	15	16	Bit map of unit codes.
26	HOUSECODE				0	0	0	0	Same as table 1.
27	LEVEL				FUNCTION				Same as Notes 1 and 2.
28	CHECKSUM								Sum of bytes 20 to 27.

X=DON'T CARE

TABLE 4 - TIMER MODE SELECTION

BYTE 20 lower nibble

D3	D2	D1	D0	MODE	EXPLANATION
1	0	0	0	NORMAL	Occurs on a weekly cycle at the same time each day, on day or days specified by byte 21 and at the time specified by bytes 22 and 23. The function and codes for event are specified by bytes 24 to 27.
1	0	0	1	SECURITY	Same as NORMAL mode except that the event time will be different each day and will be within one hour after the time specified by byte 22. (varies in a pseudo random pattern). SECURITY is only available in EVERYDAY and SPECIFIC DAYS modes, see note 5.

TABLE 4 - TIMER MODE SELECTION

BYTE 20 lower nibble

D3	D2	D1	D0	MODE	DESCRIPTION
0	1	0	0	TODAY	EVENT occurs only TODAY at the time specified by bytes 22 and 23. and will be cleared from memory at midnight TODAY.
0	0	1	0	TOMORROW	EVENT occurs only TOMORROW at the time specified by bytes 22 and 23. and will be cleared from memory at midnight TOMORROW.
0	0	0	0	CLEAR	Clears from memory, the event specified by the event number stored in bytes 18 and 19.

NOTE 5

In addition to TODAY and TOMORROW, it is suggested that the program offer the user the choice of EVERYDAY and SPECIFIC DAYS. If EVERYDAY is chosen, byte 21 should be sent as 7F HEX (all days selected). If SPECIFIC DAYS is chosen, byte 21 should indicate which days were chosen.

GRAPHICS DATA DOWNLOAD

Graphics data is stored in bytes 1024 to 1535 of the 2K x 8 RAM in the interface. Only bytes 20 and 21 of the downloaded message are stored. Each pair of bytes is assigned a number between 0 and 511 as specified by A0 to A6 in byte 18 and A7 in byte 19. D0 in byte 18 is ALWAYS '0' so these address numbers increase in steps of 2 (for graphics type and X-10 code of 256 objects). Note also that byte 19 D1 is ALWAYS "0" and D2 is ALWAYS "1". The computer should keep track of the message numbers and load new messages into vacant address locations. The contents of bytes 20 and 21 depends on the graphics approach used by the programmer (see note 6), the interface merely stores this data and will upload it to the computer upon request (see graphics upload). Byte 22 is the sum of bytes 20 and 21.

GRAPHICS DATA DOWNLOAD

BYTE	D7	D6	D5	D4	D3	D2	D1	D0	
1-16	1	1	1	1	1	1	1	1	Sync 16 X FF
17	0	0	0	0	0	0	1	1	ID3, event/graphics download.
18	A6	A5	A4	A3	A2	A1	A0	0	A0 to A7, binary number for
19	X	X	X	X	X	1	0	A7	graphics object- 256 objects.
20				GRAPHICS DATA					User RAM to define type and
21				GRAPHICS DATA					X-10 code of graphics object.
22				CHECK SUM					Sum of bytes 20 and 21.

X= DON'T CARE

Note 6
A suggested allocation for byte 20 is shown below.

BYTE 20	D7	D6	D5	D4	D3	D2	D1	D0
	1=ON 0=OFF			ICON TYPE				

FOR EXAMPLE

	D7	D6	D5	D4	D3	D2	D1	D0
ICON of a lamp shown in the ON state.	1	0	0	0	0	0	0	1
ICON of a T.V. shown in the ON state.	1	0	0	0	0	0	1	0
ICON of a coffee pot shown in the ON state.	1	0	0	0	0	0	1	1
ICON of a fan shown in the OFF state.	0	0	0	0	0	1	0	0

Byte 20 = 0 indicates a vacant ICON storage location. To clear an ICON from the Interface you need to send a graphics download with byte 20 = 0. Note, after doing this, you should also send a DOWNLOAD TIMER EVENT message with byte 20 = 0 (to clear any timed events for the removed ICON).

A suggested allocation for byte 21 is shown below.

BYTE 21	D7	D6	D5	D4	!	D3	D2	D1	D0
		HOUSECODE OF STORED ICON			! !	UNIT CODE OF STORED ICON			

REQUEST CLOCK AND BASE HOUSECODE
(Interface to Computer)

To upload the time and Base Housecode from the Interface it is first necessary to send a leading SYNC pattern of 16 X FF bytes, followed by an ID4 for the request clock and Base Housecode. See below.

BYTE	D7	D6	D5	D4	D3	D2	D1	D0	
1-16	1	1	1	1	1	1	1	1	SYNC FF X 16
17	0	0	0	0	0	1	0	0	BASE Housecode.

If the Interface receives the request correctly it will respond by uploading the clock and Base Housecode to the computer, as shown. If the request is not received correctly, no response is given.

CLOCK AND BASE HOUSECODE UPLOAD

Byte	D7	D6	D5	D4	D3	D2	D1	D0	
1-6	1	1	1	1	1	1	1	1	SYNC 6 X FF
7	0	0	0	0	0	0	0	S	status bit. *
8	0	0			MINUTES				HEX 00 to 3B (0 to 59)
9	0	0	0		HOURS				HEX 00 to 17 (0 to 23).
10	0	Sun	Sat	Fri	Thu	Wed	Tue	Mon	Bit mapped days.
11	BASE HOUSECODE				0	0	0	0	Same as table 1.
12				CHECK SUM					Sum of bytes 8 to 11.

* The STATUS bit is reset to "0" during power up of the Interface and is set to "1" by a DOWNLOAD of data from the computer (any data with byte 17 equal to ID 0
, 1, 2, or 3). The STATUS bit is used to warn the computer that the Interface has been powered down. E.G. a STATUS bit equal to "0" could tell the program to display a message such as "The Interface has been powered down and contains no data. Press Enter to continue".

REQUEST TIMER EVENTS (Interface to Computer)
To upload the timer events from the Interface it is first necessary to send a leading SYNC pattern of 16 X FF bytes, followed by an ID5, for request timer events. See below.

BYTE	D7	D6	D5	D4	D3	D2	D1	D0	
1-16	1	1	1	1	1	1	1	1	SYNC FF X 16
17	0	0	0	0	0	1	0	1	ID5, request timer events.

The Interface will respond by uploading to the computer, all of the 128 events starting with number 1 as shown below. A vacant event is represented by a single FF byte, this shortens the time for the upload. The check sum does not include these FF bytes.

TIMER EVENTS UPLOAD
EXAMPLE WHERE ONLY FIRST TWO EVENTS ARE PROGRAMMED

BYTE	D7	D6	D5	D4	D3	D2	D1	D0	
16	1	1	1	1	1	1	1	1	SYNC FF X 6.
7	0	0	0	0	0	0	0	0	Status.
8-15	EVENT NUMBER 1 AS DOWNLOADED, 8 BYTES.								
24-149	1	1	1	1	1	1	1	1	FF X 126 to indicate 126 vacant event spaces.
150				CHECK SUM					Sum of bytes 8 to 23 (FF bytes ignored).

REQUEST GRAPHICS DATA (Interface to Computer)

To upload graphics data from the Interface it is first necessary to send a leading SYNC pattern of 16 X FF bytes followed by and ID6, for request graphics data. See below.

BYTE	D7	D6	D5	D3	D2	D1	D0	
1-16	1	1	1	1	1	1	1	SYNC FF X 16
17	0	0	0	0	1	1	0	ID6, request graphics data.

The Interface will respond by uploading to the computer, all of the 256 ICONS starting with number 1, as shown on page 33. A vacant ICON space is represented by a single FF byte, this shortens the time for the upload. The check sum does not include these FF bytes.

GRAPHICS DATA UPLOAD
EXAMPLE WHERE ONLY 5 ICONS ARE PROGRAMMED

BYTE	D7	D6	D5	D4	D3	D2	D1	D0	
1-6	1	1	1	1	1	1	1	1	SYNC FF X 6.
7	0	0	0	0	0	0	0	0	Status.
8-9				ICON NUMBER 1					2 bytes.
10-11				ICON NUMBER 2					2 bytes.
12-13				ICON NUMBER 3					2 bytes.
14-15				ICON NUMBER 4					2 bytes.
16-17				ICON NUMBER 5					2 bytes.
18-268	1	1	1	1	1	1	1	1	FF X 251 to indicate 251 vacant ICON spaces.
269				CHECK SUM					Sum of bytes 8 to 17 (FF bytes ignored).

DIAGNOSTIC

The Interface has a self test diagnostic routine which is initiated by sending a leading SYNC pattern of 16 X FF bytes followed by a ID7. Upon receiving this instruction, the Interface will run a self check on it's own hardware and software (firmware). The output of the Interface (pins 3 and 4) will go low for 10 seconds as part of this test. If the check is o.k. the Interface will respond by sending the ACK with status "0". If a fault is diagnosed, the interface will Send ACK with status "1" or will not respond.

BYTE	D7	D6	D5	D4	D2	D1	D0	
1-16	1	1	1	1	1	1	1	SYNC FF X 16
17	0	0	0	0	1	1	1	ID7, initiate self test.

The above programming guide is included when you purchase the interface. I checked and it is freely distributable. And remember, if you don't want to control more than 95 modules, you don't need to create your own program (as stated previously.

If you want to read further on the X-10 system, then go to there web page at http://www.x10.com. They have information on all the products they carry, as well as ones still in the works. Also on the web page is a FAQ (Frequently Asked Questions) file that has a wealth of information on the X-10 system and compatible products. Unfortunately, the interface is no longer sold packaged with the Commodore cable and software. So, you must purchase the interface packaged with some other software and call X-10 and order the Commodore connection package separately for $25. Of course you can create a cable and copy the software (it is also freely distributable) if you wanted to. To get the interface and any modules or compatible products, Home Controls has a great selection, and there catalogue is filled with home automation equipment. X-10's and Home Controls addresses and phone numbers are as follows:

X-10 (USA) INC.
91 Ruckman Rd.
Closter, NJ 07624
Phone: 201-784-9700

Home Controls INC.
7626 Miramar Road, Suite 3300
San Diego, CA 92126-4446
Phone: 800-266-8765 or 619-693-8887

```
::::::::::d:i:s:C=:o:v:e:r:y:::::::::::::::i:s:s:u:e::3::::::::::::::::::::
\H02:::::::::::::::::::::::::H:A:R:D:W:A:R:E:::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

```
 !......The Metal Shop......!
                                                    // - The Metal Shop - //

                      ^...!..The Metal Shop..!...^
```

```
  Contributors : XmikeX  (xmikex@eyrie.stanford.edu)
                 Co-Sysop of DiamondBack BBS (305)258-5039

                 David Wood (jbevren@willowtree.com)

                 Marc-Jano Knopp and Daniel Krug
                 http://www.student.informatik.th-darmstadt.de/%7Emjk/c64.html
```

Dear Metal Shop,

I recently found a 16 KB Commodore expander for my trusty VIC-20. My glee turned to disgust after realizing that I needed the proper DIP scheme required to bank the memory into the VIC. I barely know the old '20, much less this expander. Please tell me this will be easy.... :)

RE
--

RE,

You are in luck RE because as the Good Lord would have it, I have also run into a VIC-20 expander! Ok...I didn't just "run" into it... I actively sought one out to view the new Veni Vedi Vic (Vedi Veni Vic?) demo by Marko Makela (with umlauts over the a's). :) And oh Yes, this will be easy!

Without much further ado, I decoded the DIP scheme for my 16 KB VIC-20 expansion cartridge, as follows :

CBM VIC-1111 16 KiloByte RAM expansion cartridge with eight DIP scheme
----

Expansion RAM bank 1 (8 KB)					Expansion RAM bank 2 (8 KB)			
				8 KB Mapped to :				
1	2	3	4		5	6	7	8
on	off	off	off	-> $A000-RAM/ROM4 <-	on	off	off	off
off	on	off	off	-> $6000-RAM/ROM3 <-	off	on	off	off

```
off   off   on    off   -> $4000-RAM/ROM2 <-    off   off   on   off
off   off   off   on    -> $2000-RAM/ROM1 <-    off   off   off  on
```

As is evident above, the DIP settings for each bank yield identical results so I suspect that the settings for the 8 KB CBM ram expander follow the same scheme. 8 KB chunks do keep things simple, and we can either give the VIC-20 just 8 KB of expansion memory at $2000, $4000, etc., or we can activate both banks and map them to these RAM/ROM expansion blocks within the VIC's memory map for 16 KB total. For example, whereas 8 KB ROM images are usually quite happy with 8 KB of memory at RAM/ROM4, most 16 KB rom images tend to require that RAM/ROM blocks 3 and 4 be active. We can accomplish this by mapping expansion bank 1 into $6000 and bank 2 into $a000 (or vice versa!). In this case, our DIP settings could be :

```
        1     2     3     4     5     6     7     8
        off   on    off   off   on    off   off   off

                              OR

        on    off   off   off   off   on    off   off
```

Please note that with the CBM 16k expander, we cannot map any of its memory to the 3 KB expansion area sitting at $0400-0fff (1024-4095). - XmikeX
--

[Ed. Note : The following is a response to the 1541 <-> Apple disk][
 discussion that appeared in this column, issue 2. It has been reprinted
 in its entirety with permission from the author.]

>From   : jbevren@willowtree.com
Date   : 19-JAN-1997 08:58:43.19
Subject : info on 1541/disk][transfers

First: Thank you disC=overy magazine. I've read a few zines, and only yours and C= Hacking make the bet for me.

Keep up the _good_ work!

Dear Metal Shop,

I was reading the article in the disC=overy magazine, issue two. In there was a section from a user who has an Apple][computer who wished to transfer files from the computer's disk drive to the Commodore 1541 (or was it vice-versa?).

I am commenting on the hardware aspect of the disk][drives on the Apple computer. When I was in college, my 1541 failed completely due to lightning. The main board was fine, but the read/write head was blown "open" as I discovered in the school's electronics lab. Remembering my days on the ol'][in high school, I thought of hooking a disk][assembly to the 1541 board.

I acquired a junk drive assembly from a friend, unhooked all the electronics, and re-wired the assembly to the 1541 drive's CPU board. Voila! The system ran fine, until I could come up with a low-density single-sided assembly from a PC to fit "inside" the 1541's case. It even ran fastloaders and GEOS.

Point at hand? The disk]['s mechanical assembly is functionally identical to the 1541's drive assembly. In fact, the Alps assembly was used in some Apple compatible drives (am I right?). Reading Apple disks on the 1541 will be one hell of a challenge, because of the drive's limited RAM.

However, reading Commodore's GCR format should be fairly easy if you have a little advanced knowledge of the disk][interface. All you have to do is get a modified CBM DOS binary into the Apple][, and have it read the Commodore disk. Not a trivial task mind you. When the DOS is in the Apple, the rest is easy. simply change the locations the program uses to read/write the disk, and you have it.

Hope I was helpful!

-David Wood

ps: All the disk][is (with the interface included), is a 1541 with no brains. :)
--

[Ed. Note : The text below to the end of the article is a sample of the C=
hardware repair and enhancement projects put out by Marc-Jano Knopp & Daniel
Krug on the WWW http://www.student.informatik.th-darmstadt.de/%7Emjk/c64.html
Reprinted with permission.]

SAFER SID!

Project: Protection circuit for SID
Target : C64 (all models)
Time   : 20 min
Cost   : ~1 US$
Use   : Protect the SID from overvoltage.

Cause

The SID cannot stand more than 3Vss or 1Veff, respectively.
Therefore, if you use the SID audio input, the input voltage
should be limited. The circuit below limits the input signal
to about 1.4Vss. This is done by two antiparallel silicon diodes
which cut off the signal at +/- 0.7V. The resistor limits the
current coming out of the audio source.

Ingredients

- 1 resistor 470 ohm
- 2 diodes 1N4148 (or 1N914)

Instructions

1. Build the following circuit:

```
  SID protection circuit
  ----------------------


  (5)  O---------o----------o-------XXXXX-----O  Audio in
                 |          |
                 |          |                     __
              +-----+     -----                  /  \
 <-- to SID    \   /       / \          <--   -+----+----+-
                \ /       /   \                      \__/
               -----     +-----+
                 |          |
  (2)  O---------o----------o-----------------O  GND


  XXXXX  = resistor 470ohm
  diodes = 1N4148 or 1N914
```

2. ... and stuff it in a DIN plug for the AV jack.

SPONTANEOUSLY RESETTING C-64?

Symptom: C-64 resets at random.
Target : C64 (all models) and VC20
Time   : 10 min.
Cost   : <1 US$

Cause

Voltage level floats near to forbidden zone (unstable voltage
below ~4 volts).

Fix

Pull-up resistor between pin 3 (/RESET) and pin 2 (+5V) of
user port. Value should be in the range of 4.7k to 10k,
starting with 10k.

Note

C-64s which are used industrially, should be equipped with an additional 100nF capacitor connected between pin 3 (/RESET) and GND (pin 1), which provides excellent noise immunity. The same applies to users encountering reset problems due to a long reset switch cable.

Ingredients

- resistor of 10k (or less)

Instructions

1. Open case, disconnect power plug!

2. Solder one end of resistor to pin 2 the other end to pin 3 of the user port connector. DO NOT solder the leads directly onto the contact area, instead try soldering them to the nearest visible bare point connected to the above pins.

3. Connect power plug and keep your C-64 running for several hours. See if it resets spontaneously as before; if not, close the case again.

Possible failures

- Resistor's value is too high, try 5.6k or 4.7k otherwise.

'FIX' RESTORE KEY!

```
Project: Make RESTORE key behave like all the other keys
Target : C64 (old)
Time   : 15 min.
Cost   : <1 US$
Use    : Never hear the RESTORE key singing
         'Killing me softly ...' again.
```

Extract

Replace 'NMI-capacitor' C38 by a 4.7nF one.

Cause

The value of the built in capacitor responsible for triggering the NMI via the timer 556 is too small in old C-64's (big PCB). On account of the pressure dependant resistance of the keys, pressing the RESTORE key just as softly as the other keys is not sufficient for producing a trigger signal at the timer's TRIG input. Therefore, we will replace it with a 4.7nF capacitor.

The problem described above was fixed in the new (small) PCB's and in the SX-64.

Ingredients

- 1 capacitor 4.7nF

Instructions

1. Open case, disconnect power plug!

2. Search for capacitor C38 (51pF) below the leftmost CIA. In most boards it is built into a resistor case with green base color! Its color code is: green-brown-black

3. Unsolder it and replace it with the 4.7nF one.

4. Connect power plug.

5. Switch your C-64 on. After the startup screen appeared press [STOP]-[RESTORE] just as softly as you would usually hit the other keys. Repeat that procedure a few times and if the RESTORE key seems to behave like the other keys, close the case again.

Possible failures

- Replaced the wrong capacitor. Often the capacitor is built into a resistor case.

BLACK SCREEN

Symptom: Screen remains black.
Target : C-64

Possible sources of failure
- Power supply's fuse is blown -> no power :)
- continuous reset -> no startup
- Fuse -> no 12V for VIC (old C-64)
- Voltage regulator -> no 12V, no fun (old C-64)
- CPU -> blocks phi2 or bus
- VIC -> blocks bus
- SID -> blocks bus
- PLA -> well.... :)
- 8701 -> no clock cycles

Analysis - Power supply ok?

If the power LED is not lit, check the power supply's fuse and replace it, if necessary. Next, measure the voltage directly on the power plug. If you cannot find 5VDC and 9VAC anywhere, kick the power supply and get a new one.

If you have an old C-64, check its internal fuse now.

Then check the 5V at the power jack (before the switch!) with the power supply plugged in and the C-64 switched on. It should be between 4.9 and 5.1V. If not, a defective IC might demand a current so high that the 5V level gets pulled below 4.8V. You should be able to locate the very IC by simply touching all the chips and checking for especially hot ones. Replace it.

Next, measure the 5V supply voltage between pin 7 (GND) and pin 14 (Vcc) of a 74xx chip. If the 5V show to be ok, then continue with 'Voltage regulator ok?'.

Otherwise, it is very likely that the cause for the low vol-tage level is - believe it or not, it DID happen to a lot of people - the power switch(!), so that it could be a good idea to either rock the switch several dozen times, open and clean it or to replace it completely.

Reset line HIGH?

Since it is possible that the black screen is caused by a con-tinuous reset, you should check the voltage between pin 1 (GND) and pin 3 (/RESET) of the user port. If it is below 1V, then check the reset switch, if you got one. Otherwise, check wether the output levels of the 7406 and 74LS14 (new board) match their input levels. Next, you should check the timer 556 (old board) when I managed to offer you a description for that chip :) Until then, simply replace it, it is not socketed, but cheap.

If that does not help, some other chips load the reset line so that the level is below 2.4V which causes the CPU to reset. Try a pull-up resistor (4.7k) between pin 2 (+5V) and pin 3 (/RESET) of the user port. If that does not make the blank screen vanish, check for other defective chips (which probably become very hot, see later in this document).

Voltage regulator ok?

In case you have an old C-64, check the 12V voltage regulator VR1 (7812). At the output (right pin, 2) you should measure around 12V (11.8 to 12.3V). If not, measure the input voltage (left pin, 1), the voltmeter should show about 15 to 20V. If the latter is the case, replace the VR. If the input voltage is out of range, check the 9VAC on the power plug (with the C-64 still switched on). If they prove to be ok, then the two diodes before the voltage regulator could be damaged; replace them. If you cannot find the 9VAC, make sure you have checked for the C-64's internal fuse and otherwise get another power supply.

CPU running?

Try accessing the floppy drive by typing blindly (you do not
need to close your eyes)

```
LOAD"$",8   (assuming your device id is '8')
```

If the floppy drive's motor starts spinning, you are lucky. The
processor, the CIA's and the address manager seem to work.
If the floppy does not react in any way, you probably have a
serious problem and should continue reading...

Other chips alive?

Test VIC, SID, PLA and, where applicable, the clock circuit 8701
near the VIC (not in very old PCB revisions), whether they get
extraordinaryly hot. If any of the chips gets hot, it is very
likely that they are defective and, e.g. block the bus. These
should be replaced. In C-64s with the new board, you will find
a 64-pin multifunction chip, the MMU. If the other chips are ok,
it is probably the MMU which is damaged. Alas, this MMU is prac-
tically impossible to replace, there are neither suitable
sockets nor spare chips.

Of course, every chip connected to the bus in any way could be
damaged and blocking the bus, therefore you should also check
if the CIAs, the ROMs, the RAMs, the RIMs or the RUMs (oops) are
getting extremly hot. Especially in the old C-64 board, the
various 74xx/74LSxx might be responsible for the failure.

If all else fails...

... you should consider the very low price of a used C-64 on
the free market or, if you live in a strange country with prices
over US$ 30 for a C-64 (including power supply), check for
hair cracks and dry joints.

```
::::::::::d:i:s:C=:o:v:e:r:y::::::::::::::::i:s:s:u:e::3::::::::::::::::::::
\L01::::::::::::::::::::::::::::L:E:G:A:L::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

A R T I C L E S   O F   O P E R A T I O N

Article 1 : Mission Statement

Our intent is to present useful information in order to enhance and preserve
the knowledge base of the Commodore 8-bit domain, including, but not limited
to, the Commodore 64 and Commodore 128 home computers. To this end, we shall
require that every article contain what in our discretion should be a viable
Commodore 8-bit hardware and/or software point of relevance. Likewise, each
issue should include material that can both potentially enlighten the most
saavy of users as well as the layman. We intend to complement and assist all
others engaged in similar endeavours. We believe it is of paramount concern
to stave off entropy as long as possible.

Article 2 : disC=overy Staff

The current staff of disC=overy, the Journal of the Commodore Enthusiast,
is as follows:

Editor-in-Chief	: Mike Gordillo	(s0621126@dominic.barry.edu)
Associate/Tech Editor	: Steven Judd	(sjudd@nwu.edu)
Associate/Tech Editor	: George Taylor	(aa601@chebucto.ns.ca)
Webmaster	: Ernest Stokes	(drray@eskimo.com)

disC=overy, issue 3 logo by 'WaD'

We invite any and all interested parties to join us as authors, panelists,
and staff members.

Article 3 : General Procedures

- Submission Outline -

a. Articles may range in size from 1 kilobyte and up. Approximately 15
   kilobytes of text is the preferred length, including any software present.

b. Sufficient technical content about Commodore 8-bit home computers, concerning software and/or hardware relevant to these systems, is a requirement. What constitutes a sufficient amount of 'technical content' is left to the discretion of the Editor-in-Chief and/or the review panel (see below).

- Staff Priorities -

The Editor-in-Chief shall supervise the organization of each issue in regards to grammatical and syntactical errors, flow of content, and overall layout of presentation. The Editor-in-Chief and Associate Editor shall form a review panel whose function it shall be to referee literary work which the Editor in-Chief has deemed to be of advanced technical merit. The Editor-in-Chief iand disC=overy, the Journal of the Commodore Enthusiast, shall retain copyright solely on the unique and particular presentation of its included body of literary work in its entirety. Authors shall retain all copyrights and responsibilities with regards to the content of their particular literary work. Authors shall be required to submit their works to the Editor-in-Chief approximately two weeks prior to publication.

Article 4 : Peer Review

To the best of our knowledge, disC=overy shall be the first Commodore 8-bit journal with a review panel dedicated to uphold the technical integrity and legitimacy of its content. The Editor-in-Chief and the Associate Editor shall be responsible for the formation of the panel. The appointed panelists shall have the option of anonymity if desired. The panel shall review works primarily for technical merit if the Editor-in-Chief and the Associate Editor deem it necessary. Authors may be asked to modify their works in accordance with the panel's recommendations. The Editor-in-Chief shall have final discretion regarding all such "refereed" articles.

Article 5 : Distribution

Although we welcome open distribution by non-commercial organizations, there are currently four "official" distribution channels available to interested parties. This journal may be obtained by directly mailing the Editor-in-Chief or via the World Wide Web at http://www.eskimo.com/~drray/discovery.html and at FTP site : ftp.eskimo.com - directory /u/t/tpinfo/C64/Magazines/discovery A "snail" mail disk copy of any issue of this journal may also be obtained from Arkanix Labs at the following physical address :

disC=overy, c/o Jon Mines
Arkanix Labs
17730 15th Ave NE Suite #229
Shoreline, WA 98155

This source should follow the distribution guidelines as listed in Article 6 below. However, specific questions concerning billing, etc., should be directed at them.

Several versions of this journal may be available for your convenience, please check with the aforementioned sources.

Article 6 : Disclaimers

The Editor-in-Chief and disC=overy, the Journal of the Commodore Enthusiast, retain all copyrights regarding the presentation of its articles. Authors retain all copyrights on their specific articles in and of themselves, regarding the full legal responsibility concerning the originality of their works and its contents.

The Editor-in-Chief and disC=overy, the Journal of the Commodore Enthusiast, grants the reader an exclusive license to redistribute each issue in its entirety without modification or omission under the following additional stipulations:

- If distribution involves physical media and is part of a commercial, not-for-profit, or PD distribution, the maximum allowable monetary charge shall not exceed $4 1996 United States Dollars per issue unless more than one issue is distributed on a single media item (i.e., two or more issues on one disk), in which case maximum allowable charge shall not exceed $4 1996 United States Dollars per media item. All dollar values given assume shipping costs are -included- as part of the maximum allowable charge.

- If distribution involves non-physical media and is part of a

```
::::::::::d:i:s:C=:o:v:e:r:y::::::::::::::::i:s:s:u:e::3::::::::::::::::::::
\END::::::::::::::::::::::::::March 26, 1997:::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```