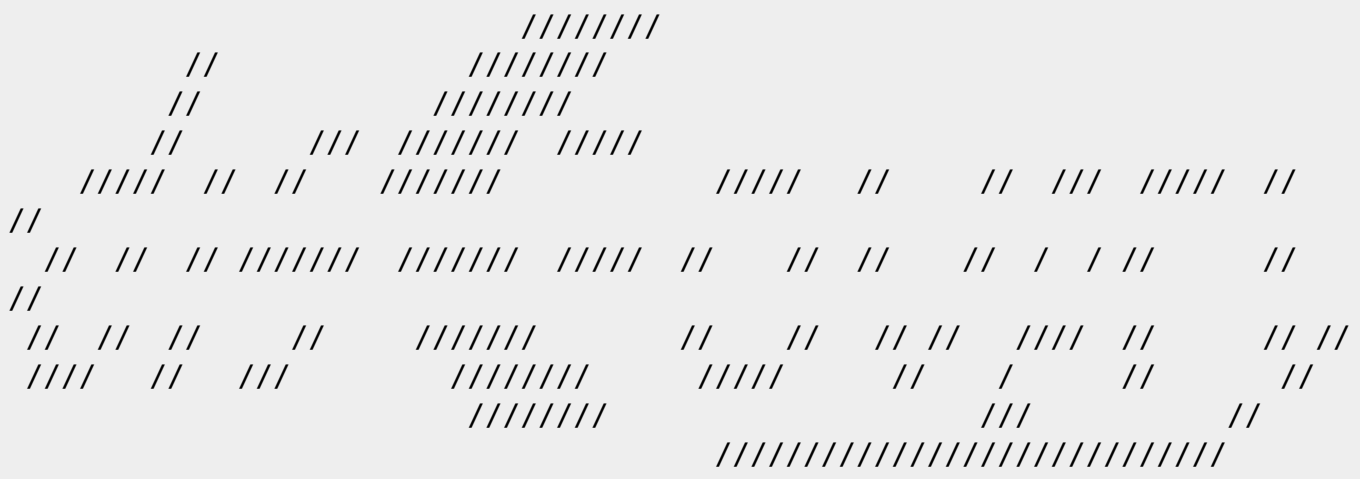

The Journal of the Commodore Enthusiast

I s s u e 1 : May 17, 1996

P R E A M B L E

We greet you to the first issue of disC=overy, the Journal of the Commodore Enthusiast. Our inspiration for launching this work derives from you, the ones who still hold our beloved 8 bit machines in high regard and respect. In honor of your committment to these classic platforms we have pledged ourselves to assemble this entire journal on modest C64 and C128 systems. It is our sincerest hope that you will find our efforts to be of interest and special joy. We thank you from the bottom of our hearts and look forward
to forging a solid productive relationship with all of you.

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

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 Editor : Steven Judd    (judd@merle.acns.nwu.edu)
        Webmaster        : Ernest Stokes  (drray@eskimo.com)

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


Article 3 : General Procedures

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 and/or social merit. The
Editor
in-Chief and 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 two "secure" 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

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 commercial, not-for-profit, or PD distribution, the maximum allowable charge shall be limited to the actual cost of the distribution, whether said cost be in the form of telephony or other electronic means.

It is understood that distribution denotes acceptance of the terms listed and
that under no condition shall any particular party claim copyright or public domain status to disC=overy, the Journal of the Commodore Enthusiast, in its entirety.

The Editor-in-Chief and disC=overy, the Journal of the Commodore Enthusiast, reserve the right to modify any and all portions of the Preamble and the Articles of Operation.

::::::::::d:i:s:C=:o:v:e:r:y::::::::::::::::i:s:s:u:e::1::::::::::::::::
::
::::::::::::::::::::T A B L E O F C O N T E N T

```
S:::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::


-Software Section-

/S01 - "Explorations on IFLI"
$d000   by Adrian Gonzalez and Mike Gordillo

/S02 - "TRI-FLI: A new video mode 'abrewing?"
$d000   by George Taylor and Mike Gordillo

/S03 - "Heaven in the net, an unedited excerpt of IRC on #c-64"
$d000   by Mike Gordillo

/S04 - "A complete dissection of Gfx-Zone"
$d000   by "XmikeX"

/S05 - "A Beginner's Guide to the JCH Editor V2.53, NewPlayer V14.G0"
$d400   by Sean M. Pappalardo

/S06 - "An inside look at MODplay 128"
$d400   by Nate Dannenberg and Mike Gordillo

/S07 - "Some preliminary data on VDC timing"
$d600   by Steven L. Judd

/S08 - "Software analysis and reconstructive therapy, a historical view
$dd00   on 'cracking'"
        by Pontus Berg

/S09 - "A Quick Overview of CP/M"
0100h   by Mike Gordillo

/S10 - "The BIOS-R62a/ZPM3/ZCCP Commodore 128 CP/M 3.0+ Upgrade Package
0100h   and a bunch load of utilities!"
        by Mike Gordillo


-Hardware Section-

/H01 - "BEYOND STEREO - The POWER-SID Model 2400 : It May Not be THX but
        it's Taking the Commodore One Dimension Closer"
        by Shaun Halstead

/H02 - "The 8 bit Modplay 128 Board"
        by Nate Dannenberg

/H03 - "Upgrading your C128's VDC memory to 64K"
        by Mike Gordillo
```

```
/H04 - "The Metal Shop"
        with SMS Mike Eglestone


::::::::::::d:i:s:C=:o:v:e:r:y:::::::::::::::::i:s:s:u:e::1:::::::::::::::::
::
/S01::$d000::::::::::::::::::S O F T W A R
E::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
```

Explorations on IFLI

by Adrian Gonzalez and Mike Gordillo

Adrian Gonzalez is a true Commodorian, a rare breed in his native country of Mexico. He has spent the past year or so converting .gif and .jpeg files into the IFLI format on the C64. His endeavours have allowed many a C64 owner to enjoy high quality images.

> Adrian, before we start, can you give us some background on what IFLI is and
> how it achieves such stunning images?

Sure thing. A little multicolor bitmap mode and FLI mode background is in order too. Images in the 'regular' multicolor bitmap mode (MCBM from now on) are divided in cells or blocks that are 4 pixels wide and 8 pixels tall. The screen is split into 40 columns of these blocks horizontally and 25 vertically, giving a total of 1000 4x8 pixel cells or blocks. A pixel in one of these blocks can have any one of 3 colors common to the entire block, or the background color, common to the entire screen. A FLI picture is similar to a regular MCBM picture in that it has the same resolution (160x200), however, it is more flexible in terms of how many colors you can use in each 4x8 character block. This flexibility is achieved through complex timing tricks which I'd rather explain in a future article, but I'll give a very brief overview of what they make the VIC-II do. Basically, the trick is to fool the VIC chip into fetching the screen memory data on every rasterline. This screen memory data is responsible for 2 of the 4 colors available on each 4x8 character block

```
in
MCBM (bit pairs '01' and '10').  So this basically divides each 4x8 cell of
MCBM into eight 4x1 cells, making the VIC chip fetch a different pair of
colors
from screen memory for each rasterline (each 4x1 cell).  This trick gets rid
of
most of the restrictions that MCBM imposes, since now 2 of the 4 pixels in
every cell can have any one of the 16 available colors.  Chances are the
colors
needed for the other two pixels will be available from either screen memory,
color memory or the background color.  This flexibility, of course, does not
come without a price:  storage.  This technique multiplies the amount of
memory
needed for screen data times 8, adding 7000 bytes when you compare with a
standard koala paint file.  Fortunately, the 8000 bytes needed for screen
memory and the 8000 for the bitmap fit just right on one 16k bank (remember
the
VIC can only 'see' 16k at a time).

    If you are new to these software screen modes, I recommend you read the
previous part again, especially because once you get the grasp of FLI mode,
IFLI is a snap.  IFLI mode is basically two FLI pictures being rapidly
alternated to give the illusion of more colors.  So where does the added
resolution come in?  Thanks to the VIC-II's hardware, it is possible to
shift
the screen up to 7 hi-res pixels horizontally, even when in MCBM.  The trick
in
IFLI is to display one FLI picture for an entire screen refresh, then
display
the second FLI picture shifted one hi-res pixel to the right on the next
redraw.  The effects of this are not so obvious, so I'll illustrate with an
example.  Suppose you have the first line of FLI picture A and FLI picture B
and it looks like this:

First line of FLI picture A:
Ü  K  Ü  G  Ü  Y  Ü  O  Ü . . .

First line of FLI picture B:
Ü  G  Ü  G  Ü  Y  Ü  O  Ü . . .

When you alternate them, shifting picture B one hi-res pixel to the right
you
get:
Ü  K  Ü  G  Ü  Y  Ü  O  Ü. . .
   Ü  G  Ü  G  Ü  Y  Ü  O  Ü
------------------------------
ÜK ÜKGÜG ÜG ÜGYÜY ÜYOÜO Ü. . .

Where:
K = Black pixel
G = Green pixel
```

```
Y = Yellow pixel
O = Orange pixel

The resulting line has pixels that are as wide as high resolution pixels,
and
that may have colors that are combinations of the c64's 16 standard colors.
It
is evident from this example that IFLI's strengths lie in reproducing images
with smooth color shades.  The downside of IFLI is that it flickers, and
depending on the colors that are being alternated, the flicker can go from
barely noticeable to a stroboscopic light show.  With this in mind, however,
this display mode can produce some of the most stunning images that have
ever
been displayed on our beloved C64.  After this brief introduction I hope the
interview will make a little bit more sense, so let's get on with it.


> Adrian, I know you do the bulk of your conversions on other platforms for
> the sake of expiediency.  However, I'm more generally interested in how
> the pictures get down to the C64 in terms of the actual display on the
C64.

Well, I start out by doing careful color analysis on my Amiga.  This
involves
pre-processing the images in programs such as The Art Department
Professional,
in order to adjust the brightness, contrast, color and size of the source
images.  After that, I feed the images to a conversion utility I wrote that
tries (as best as possible) to map the colors from the original image to
combinations of the 16 colors of the c64, taking many factors into
consideration, such as flicker.  The conversion program has several
settings,
which have to do mostly with dithering and flicker reduction.  It is not
always
perfect, and the problem of IFLI flicker is never truly defeated, but I'm
generally satisfied with the results.


> Yes, they are beautiful conversions, but I could never get them to display
> with any other IFLI viewer.

Since there is no standard data format for IFLI pictures I had to come up
with
my own.  My ifli's load at $2000, then the code moves them to the higher
memory
and depending on how big they are, of course, they get depacked accordingly.


>Argh, they are packed?

Of course they are packed.  When was the last time you saw two of them
```

```
having
the same size? :)


> What kind of packer did you use?

They are packed using a very simple RLE routine.  If you want you can use
the
routines in my viewer to depack them for you.  They'll set everything up in
the
right place. (e.g. $4000-7fff, $c000-$ffff and $d800-$dbff).  If anybody is
interested in these routines or the source code for the viewer, feel free to
mail me and I'll send them your way.


> Why did you split the data over those addresses?  Is it safe to assume
that
> IFLI data is composed of two 16 kbyte blocks for pic data and then comes
the
> color info at $d800 on up?

Yeah, you can look at it that way.  It's actually two 8k blocks containing
the
bitmaps and two 8k blocks containing the screen memory.


> Ah, I see.  There is one FLI pic portion in each 8k bitmap and one 1k
block
for color memory that is common to both images.

Yes, but you know, it might be interesting if it were possible to change
color
memory as well.  It would give us a slight boost in the number of apparent
colors.


> Could you explain exactly how, and wouldn't any further memory tweaking
> complicate matters more?  Maybe you would need to not use the entire
screen?

Oh, just mumbling out loud :).  In practice there is no time to change color
memory on the fly.  Also, it doesn't matter if you do IFLI's smaller than
the
entire screen (say 1/3 screen IFLI) because you still have data scattered
all
over the same ranges I described earlier.  The IFLI data must be placed at
certain spots in memory so that the VIC-II chip can properly fetch it.


> Ok, I understand, but this 1/3 IFLI screen proposal intrigues me greatly.
> I hear "less than full-screen" IFLI's may be doubled-over onto themselves,
```

> thereby giving even more colors or perhaps leading to run-time IFLI
> animations.

This is a possibility, but you would have even more flicker than regular IFLI.
I'm not even sure if the logistics involved will allow even a 1/3 IFLI to be doubled over.

> Well, if the CPU proves too slow, what about using the speedy REU to do the
> job?

Interesting, the problem is that since I'm not lucky enough to own one, I do not know how much overhead it takes to set up the REU for multiple transfers.
For example, if I wanted to transfer 512 bytes of screen memory, then reprogram
the REU to transfer another 512 bytes, I would need to know how many cycles it
would take for each request I sent to the REU and then fit it into the IFLI code. As I hinted at earlier, IFLI data (except for the bitmaps of course) is
scattered in little chunks.

> I purloined some standard pre-code from George Taylor. It sets up the REU
> transfer with regards to screen memory, as follows:
>
```
> lda #0
> sta control ; to make sure both addresses are counted up
> lda #<$0400
> sta c64base
> lda #>$0400
> sta c64base + 1
> lda #0
> sta reubase
> sta reubase + 1
> sta reubase + 2
> lda #<$0400
> sta translen
> lda #>$0400
> sta translen + 1
> lda #%10010000;  c64 -> REU with immediate execution
> sta command
```
>
> Total cycle count is around 60 cycles in this case, but if you shave off
> some formalities (direction, transfer length) you can cut it to 30 cycles,
> as follows:
>
```
> lda #<any64addy
```

```
> sta c64base
> lda #>any64addy
> sta c64basehi
> lda #<anyReuaddy
> sta reubase
> lda #>anyReuaddy
> sta reubasehi
> lda #$91
> sta command


> So if you implemented a set up time of 30 cycles per transfer, would this
> allow some IFLI doubling or animations?

Hmm, you need 8 requests for screen mem + 1 for color + 2 bitmaps.  So, 30
cycles just for the setup and then 1 byte per cycle transfer... that's quite
good.  I think even my most optimized copy routine for internal memory moves
would not match this, at least for large transfers (you've gotta love DMA :-
).


> But even with 60 cycles or less, could the reu copy memory much faster
with
> all the little bits and pieces involved.

Yeah, either that or use 2 mhz mode on a 128, which would not be something
I'd
like to do.  I think the REU is a better vehicle for doubled-IFLI or even
IFLI
animations, although I do not think full-screen IFLI's could be coaxed into
it.


> Well, just shrink an IFLI pic to the dimensions of a sprite and double it
or
> animate it over several times :)

Come to think of it, pc users watch video in postage stamp-sized windows :).
But there are other factors to consider though, such as the effect of the
picture alternation on the animation.  The IFLI color effect could be
lessened
by an animation with a high frame rate.  It would be a very good experiment,
though.  Who knows, maybe with the upcoming 20mhz boards we could even have
an
MPEG player for the c64 :).  As for me, all I need now is somebody to donate
their REU in the name of IFLI research :-).

--
For more information, gripes, etc, Mr. Adrian Gonzalez may be reached at the
following internet addresses: al170866@academ01.mty.itesm.mx
                              agonzalez@nlaredo.globalpc.net
```

```
/S02::$d000::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
```

TRI-FLI : A new video mode 'abrewing?

by George Taylor and Mike Gordillo

George Taylor can be described as one of the few hard-core Commodore 64 purist coders. His mission is to advance the applicable knowledge base and utility of the C64 by bringing to bear a deep understanding of the hardware and software behind it. It is often said that chance favors the prepared mind, but in George's case, the prepared mind leaves nothing to chance.

> George, I am very curious about your proposed tri-fli technique. I hear
> that it will "go beyond" what is currently available in terms of VIC-II
> video on the 64. I would be honored if you would take the time to clarify
> things for me and our readers.

Basically, I have an idea that allows no flicker unlike true IFLI, while still
increasing the amount of perceived colors on screen. Right now I am testing the idea with 4 interlaced hires (not FLI) screens. Unfortunately, the colors
are so close together I can hardly tell them apart. In other words, I get 4096 colors but I can't even see them all. When I scale the project up, and work out the problems, it may evolve into what I call "tri-fli".

> Sounds nifty. Let's backtrack a little though. Your current testbed is
> 4 interlaced hires (320*200?) screens. Just HOW are you getting even this
> set up to NOT flicker?

You seem to think this part of it is difficult, Mike. :)

> Yes, I do -sans flicker-. I'll be amazed if you can scale it up to the FLI
> domain where timing and other concerns label the process to be difficult.

Even if they were FLI, there would be no problem with timing..

> Over my head.. fli =! timing?

Generally speaking, interlace switching occurs in the borders, so you get lots of time even on an NTSC machine. The idea is that -one- FLI screen takes careful timing, but interlacing several FLI screens is NO problem!

> So your technique is basically IFLI revisited?

```
Hmm... well..yes and no.  I should clarify a few things :).

> Please do, Professor Taylor :).

Multicolor mode is a bitmap + a color map, FLI is a bitmap + * color maps.

> I'm with you so far.

Normally, each color map byte affects an -8x8- area.  In FLI, the color map
is
changed each line with a raster.  Thus, 8 color maps are used in 8 lines,
and
each byte of each color map now affects an -8x1- area.  This is why FLI
gives
a better placement of colors.  However, there is more to this.  For example,
to switch the color maps in fli requires a trick, otherwise FLI would be
easy
and not need any special timing.  This trick is called a DMA retrigger
(among
many other names for it) and it is necessary because the VIC-II video chip
normally only reads where the color map is every 8 lines.  Therefore, if you
switch the color map WITHOUT the DMA retrigger, nothing would happen.  The
retrigger requires a certain POKE if you will at an -exact- moment in time
on
each line in order to function.  This is where most of the whines about FLI
coding arise.  Please note that the DMA retrigger takes up 43 cycles on each
line it occurs, thereby adding to the timing woes.  FLI is just four lines
of code that are tricky to time :).

> Ouch, but what exactly does the DMA retrigger do?  It sounds like a sermon
> into "bad-lines" theology would be required to explain it :).

Hmm.. I can explain DMA later someday, somewhere :).  For now all you need
to
know is that it is a trick to cause the color map location to be read on
each
scan line, instead of every 8 scan lines.

> Hey, a layman reponse!  I can deal with it.

There's no point in being a beanie head if you can't interface with normal
types, Mike :))))).  I will take this opportunity to explain one pecularity
about the DMA retrigger that will hopefully enlighten the hard-core as well
as the laymen.  When people learned of this DMA trick, they found that the
first three bytes of the FLI picture were seen as garbage.  To compensate,
they left the left columns of the FLI as blank and to make the picture
symmetrical, the same was done to the right columns.  Thus, FLI pics are
usually cropped significantly.  I figured out just why the garbage bytes
occur and by the same token, how to avoid cropping FLI pics in the future.

Very simply, the colors shown during those first three bytes normally come
```

from color memory, but due to a bug they are actually coming from the opcode being executed at that moment! Therefore, to fix the colors, you just need to write opcodes that represent the proper colors. I looked at it, and there
is code you can write to make all 16 colors.

> Wow...that is nice detective work.

I'm sure somebody must have figured this out before, but I do not see it in the software I currently have. The problem is that most tricks are discovered
by experiment and accident, few people actually know why they work.

> The C= community tends to echo that.

This is very true. It's due to talking to many others that I understand a lot of what is going on today. I put my knowledge and that of others and try to push the limits on this old tank (64) :). Let me tell you, it is as complicated as any computer, even more so. In order to fully understand the 64, you would have to study it as if it were the culmination of an advanced engineering degree.

> I understand completely. Without the innovations pursued by a horde of
> enthusiasts we would not be having this conversation today. But just how
> are the advancements of the past being intergrated or improved upon in
> your current project?

Ok.. I wanted to finish the point about timing so you understand why even four screens is no big deal. Let's take a look at basic IFLI first and then contrast it with what I am trying to accomplish. IFLI is two screens of FLI,
but alternating with each other like a continuous animation. Frame B follows
frame A follows frame B follows frame A, etc., in an endless loop. The result
is that the two FLI pics switch places with each other so rapidly that they seem to blend together to the human eye. Therefore an apparent increase in the number of colors is the prime result. The standard IFLI technique is fast enough to do this but not fast enough to eliminate flickering as the two pics are flipped. Please remember from before that -each- FLI screen requires careful timing for the FLI/DMA stuff but this is completely unique and independent to that one particular FLI screen. The other FLI screen is just the same thing. In other words, setting up each FLI is tricky, but alternating them is not a problem. No more, no less code or timing, just different screens, ok?

> Yup, understood.

Ok.. Now, regardless of what textbooks say about the persistance of vision being 1/30 of a second, you can see much higher flicker rates and therefore my idea of using more than two screens should flicker quite a bit!

```
> Well that depends, George.  The resolution of the object and how "bright"
> the object is to begin with has to be a paramount concern in regards to
> flicker reduction.  The retina has three types of neurons and one of those
> types is responsible for "edge-assessment" of objects.  This is where
> brightness (contrast, etc) and flicker-effects are born in addition to
> persistance of vision arguments.

Exactly, I use two ideas to reduce flicker, and they work.  One idea is just
what you hinted at, to reduce contrast between colors.  Thus I choose colors
that have the same brightness but different tints, so only the color is
changing.

> The other idea is ?

[...long pause...]

This one is hard to explain.  Imagine I am flickering between two colors, A
and B.  Typically, a standard IFLI pic will have them like this:

For colors A and B

        frame 1: AAAAAAA  frame 2: BBBBBB

but I do this:

        frame 1: ABABABA  frame 2: BABABA

Think of it as marquee lights.  The even and odd lights turn on and off.

> Ah ha!  You are decreasing the "granularity" of your interlacing on the
> temporal axis ! ! !

Exactly!!!  You understand :))))))))))  This technique has even more
de-flickering power than choosing colors of the same brightness, and
together
both methods work even better.

Now that we have settled that, my idea for "tri-fli" is very simple.  If I
can
make IFLI -not- flicker, then I should be able to squeeze in another screen.
I am currently using a testbed using four hires (not FLI, IFLI) screens
because
I can more easily experiment upwards and downwards and see how much flicker
I
am getting.

> Ok, so you have set up a baseline with four hires screens... and?

I haven't seen any flicker yet.. and this is mixing four colors in real-
time.
But let me first specify -exactly- how to produce non flickering colors.
```

```
Definitions:
bitplane 0: the screen buffer which is shown on frame 0,2,4...
bitplane 1: the screen buffer which is shown on frame 1,3,5..
1: indicates foreground color in a 2 color mode
0: indicates background color in a 2 color mode

flashcolor: a color which is not one of the standard 16 colors, produced
by mixing 2 of the standard colors with this technique
palette: consists of n standard color and n-1 flash color

3 color hires mode
------------------
Screen buffer setup:
two frames are initialized, such that the bitmaps contain 0 (all
background color).  The color assignment is such that both background
colors (1 per buffer) are the same.  The other 2 foreground colors are
the same also.
An example would be black+white for each buffer.
To plot color 0 (black):
bp0: 00000000
bp1: 00000000
to plot color 1 (medium grey, flash color of black+white):
bp0: 01010101
bp1: 10101010
to plot color 2 (white):
bp0: 11111111
bp1: 11111111

7 color multi mode
------------------
both buffers have the same color maps
An example would be black+dark grey+light grey+white.
color 0 (black):
bp0: 0000
bp1: 0000
color 1 (dark dark grey, flash color of dark grey+black):
bp0: 0101
bp1: 1010
color 2 (dark grey):
bp0: 1111
bp1: 1111
color 3 (medium grey, flash color of dark grey+light grey):
bp0: 1212
bp1: 2121
color 4 (light grey):
bp0: 2222
bp1: 2222
color 5 (light light grey, flash color of light grey+white):
bp0: 2323
bp1: 3232
color 6 (white):
```

```
bp0: 3333
bp1: 3333

Note: although there are other combinations, such as:
medium grey, flash color of white+black
I do not reccomend them, as they produce nearly the same result
yet with extra flicker.  Flash colors should only be made in the
nearest combinations of luminance.  Colors are mixed nearly 50%/50%
but not exactly.
The wide pixels of multicolor mode increase flicker.
Flicker is seen most at the edges of a flashcolor where the dither
pattern is broken, and also when your eyes move.  The sensors of
your eye which detect movement will see the trailing patterns
caused by the low frame rate of the effect, and this breaks
the dither as well as lets you see the flicker.

What about a four color hires mode?

You would have 1 standard and 3 flashcolors, yet you can't dither
them properly.  For example color 1 would require:
bp0: 01010101
bp1: 10101010
with bp0 colors=black+medium grey, bp1 colors=black+light grey.
The result will be :
abababab
where a= dark grey, flash color of black+medium grey,
b=dark medium grey, flash color of black+light grey.
So you cannot create a constant shade, only a normal dithered one.
You could create a constant shading with:
bp0: 00000000
bp1: 11111111

Yet remember that this combination is not 100% flicker free.  An alternative
is in modes where you can change the palette at some other resolution,
for example in hires IFLI you could draw a constant shade of flashcolor
plus a 2 standard shades in any 8x1 area.  For example,
a byte with one half flash color and one half constant color is:
bp0: 01010011 (1=white, 0=black)
bp1: 01011100 (1=black, 0=white)
to make this more clear:
bp0: bWbWbbWW
bp1: WbWbbbWW (b=black, W=white)
with this result:
11110022 (0=black, 1=medium grey, 2=white)
It is possible to have a horizontal resolution of changing shades
averaging 8/3 pixels, for the example the above byte could have
been
00011122 (0=black, 1=dark dark grey, 2=dark grey)
and then:
01112222 (0=dark medium grey, 1=medium grey, 2=light medium grey)
so you can make shading bands of up to 9 shades.
```

```
> But how effective is the mixing?  Does blue + green = bluegreen?

As I stated earlier, I can now get 4096 colors but they are somewhat
useless.
The colors are only mixtures of the original 16 color and I can hardly tell
the
difference between blue-blue green and blue-green or green-green blue.
Right
now, I think the best use would be for grey shades.

> Hmm.. You may have to bite the bullet and introduce colors that vary a
little
> in brightness?

Well, the point of my whole idea is to eventually make standard IFLI to be
non-flickerable.  If that occurs successfully, I should then be able to
interlace it yet again in alternative fashion and be left with minor flicker
but with many more colors :).  We shall see...

--
For further information, gripes, etc., Mr. George Taylor may be reached via
email at the following internet address:  aa601@ccn.cs.dal.ca


/S03::$d000:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::



           Heaven in the net, an unedited excerpt of IRC on #c-64

                            by Mike Gordillo



As a self-proclaimed "demo freak", the following transcript (largely
unedited)
represents one of the most interesting discussions concerning C-64 that I
have
ever witnessed on IRC (Internet Relay Chat) or any other venue.  I present
this to the reader in the hopes of encouraging further participation and
patronage of IRC channel #c-64.  We begin in the middle of a dissection of
VIC chip internals by Firefoot.

> Can anything go on system-wise when a bad line is being "serviced" so to
> speak?

<firefoot> "BAD" lines are just another way of specifying the lines where
the
           VIC steals 40 cycles from the CPU to do a screen refresh.  You
can
```

```
           delay the bad lines, and push the whole screen down (FLD).  You
can
           force them to occur every line (FLI).  You can turn them off
           (blanking the screen).  You can move them horizontally (VSP).

<firefoot> The CPU isn't halted, it is busy helping the VIC chip out.
           Any code that you are executing is "halted".

<Waveform> I never understood VSP?

<Waveform> I thought the VIC brought the CPU off-line so it could sweep in
all
           the data for the bitmap in those 40 cycles (for the next 8
lines).

<firefoot> No, nothing can go on system-wise when a bad line is being
serviced,
           not in terms of the CPU anyway.

<firefoot> Wave - I guess that would be functionally identical.  I guess it
           depends on whether or not you consider the bus part of the cpu.

<Waveform> The VIC reads its data independent of the CPU, but...I still
never
           understood VSP. =)

> Yeah, but the CPU and the VIC chip can't share the bus at once.

<firefoot> I never understood VSP well. There is something about how tricky
           stuff with $d011 can cause the VIC to think it has started a new
           scanline, when, infact, it has not.

> Well..explain FLD for me then, Firefoot. :)

<Waveform> FLD is easy but FLI is not as easy and VSP scares me.

<firefoot> I've coded VSP, but never understood it either.  However I do
           understand FLD.  You just keep playing with $d011 (every scan
           line) so that the VIC keeps thinking that the *NEXT* scan line is
           the one where it is supposed to do the refresh of screen memory.
           when you stop doing this, it starts drawing the screen where it
           left off.

<Waveform> Exactly, FLD is very easy to understand but what about VSP?

<firefoot> Actually, I stumbled across VSP when coding FLI.

<Waveform> FLI is essentially the same thing as FLD. Except that instead of
           making the VIC think the next scan line is where its supposed to
           draw the screen, FLI makes the VIC think the current line is
           the one to draw on - but you do it on every line.
```

```
<firefoot> Well, wave, think of VSP as the same thing as FLD except instead
           of pushing the screen down scan lines, you push it across cycles.
           Also, try changing the delay at the beginning of an FLI routine,
           and you will see the screen shift over... voila, vsp!

<Waveform> Well, I just recently got a stable raster (double interrupt
style)
           so I haven't actually done anything that specific (FLI for
example)
           though FLD is very forgiving. You can do nifty FLD with virtually
           no timing at all.

> No timing for FLD?  ok... Why am *I* having such a hard time putting
> sprites over FLD then!

<firefoot> FLI can be made very forgiving as well (no stable raster needed).

<Waveform> Firefoot: It actually uses the CPU's Phi cycle as well as the VIC
           Phi cycle.

<Waveform> Fire: That is probably what you are remembering.

<firefoot> Well, with FLI it *is* nice to use a stable interrupt.
           I use the double raster method as well, always seemed the
           cleanest to me.

<firefoot> Oh, it is not easy to put sprites over FLD.  You have to make
sure
           that your delay each line is exactly correct.  very weird code...
           I did that and sprite over FLI with almost the same routine.
ugh.

<firefoot> Wave: That's *exactly* what I am remembering.

<Waveform> Fire: the phi thing?

> When the FLD bounces down...there -seems- to be a screen area behind it!

<firefoot> Mike, basically what I did is constructed a section of unrolled
           FLI/FLD code, and played with the delay instructions each time
           I moved the sprites so that the timing was always correct.

<firefoot> Wave, yes, the phi thing.  From the appendix about the vic chip.
           It was very informative.

> Ok...I've heard these $xfff addies pop up over and over and I guess they
> explains why the some of my pics have those lines behind the FLD!

<firefoot> Mike, the area behind the FLD is taken from the last byte of the
           video bank ($3fff, $7fff, $bfff, or $ffff).  It is also what you
           see when you open up the borders.
```

```
> Gawd, those annoying lines... I couldn't figure out where they were coming
> from.

<Waveform> Make VICBASE+$3fff equal to 0... or is it $FF?... and the lines
           will vanish.

> I also ran a few old demos..and someone (tfo?) was blabbing about how this
> border was opened up via FLD and the other one wasn't..etc., etc?  How
does
> FLD open up a border?

<[Style]> Is it true it becomes +$x9ff when extended color mode is on?

<firefoot> Style, I have never heard that, but I have never used extended
color
           mode while doing any of those.

<firefoot> Mike, the "fld opening up the side borders" thing basically uses
           the stable raster created with the fld to open the side borders.
           easier that other methods, like the double raster method, but
           sloppy, in my opinion.  besides, you can't get text or gfx....

> Firefoot, that is all well and good..but "How do you open up the nice
> sideborders?...period"... Apparently, you are telling me that the FLD
> is used as an "index" in this case.

<firefoot> Oh... well... Do you know how to open the top/bottom borders (the
           theory behind it)?

<Waveform> Well, if you time it right, you make the screen 38 columns
instead
           of 40 right at the last cycle on the raster.

> Only thing I know how to do is 38/40 column it... :D

<Waveform> Then on the next cycle, the vic thinks it has already started
           displaying the border... so it doesn't start.

<[Style]> Which location is the gfx behind an open side border???

<CRoth> Waveform:  I've tried to do that, failed miserably.

<Waveform> Style: VICBASE + $3fff

<Waveform> Anytime you open a border or open screen space and the vic
doesn't
           have normal info to fill it will it takes the last byte in its
           address space and sticks it in there.

<firefoot> Croth: Did you make a stable raster first?  Without that you will
           fail miserably.
```

```
<CRoth> Firefoot:  I couldn't figure it out.  Had a friend of mine explain
it
        to me, he was a genius when it came to that.  :)

> What about TWO FLD's?  I could swear I have seen one FLD bounce and then
> another bounce the screen behind it!  Is this lunacy on my part?

<Waveform> No it isn't, you stop one FLD...The VIC starts drawing...then
           later on, you start another FLD... You get two open spaces on
           your screen!

<[Style]> And if you can be bothered, you can make a FLD on every line &
slice
          the picture up :D

<Waveform> FLD make the whole screen data move down and up. You can achieve
           similar effects with sprites... just make all the Y coords go
           up and down.

> Ok..can you limit how WIDE the FLD is ?  Or is it always full-screen?

<Waveform> You can't FLD on every line... can you? doesn't that pooch your
pic?
           I've seen every eight lines... but every line?!?!  I thought that
           if you FLD after the VIC draws on the next line, you get a
STRETCH,
           don't you?

<[Style]> Wave, check out "Finely Sliced" by Christopher Jam.

<firefoot> Mike, I think the FLD is always fullscreen wide.  I can't see how
           it wouldn't be.

<firefoot> Same here Wave, I've only seen every 8 lines as well. Perhaps you
           could do some sort of weird cross between FLI and FLD to get
every
           line.  Of course, the timing would be most annoying.

<firefoot> Wave, I've seen that, but it seems to "stretch" every two lines?
           like that one arson demo, forget the name, and the one foe demo,
           and a few others as well.

<firefoot> [Style], is "Finely Sliced" NTSC fixed?

<Waveform> I think I accidentally did a stretch when trying to code FLD and
           it seemed like it did every two lines as well...weird.

> Indulge me, what is a "stretch" ?

<[Style]> Maybe it is every two lines.. Its been a while since I've seen it.
          I think Albion made a demo called ache! that did pretty much the
```

```
          same thing.

<[Style]> Fire: I dont know if it is NTSC fixed...

<firefoot> Mike, a stretch is an FLD with some slight modifications so that
it
           "stretches" the screen data.

[---end of transcript---]


/S04::$d000::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::


                    A complete dissection of Gfx-Zone

                                by XmikeX


The Graphics-Zone demo was released August 31, 1995.  It was received with
open arms and despite its simplicity, its unabashed unorthodoxy allowed it
to
reach the third highest spot in the Dec95. NTSC demo ballots. Gfx-Zone and
the other two files included in the gfx-zone.sfx package were the
culmination
of ten days of self-taught assembly programming.  I could not have done it
however without the guidance obtained from the following sources;

- Coder's World 1,2, and 3 by 'Wrongway and The Phantom', FOE Press :)

- 128 Machine Language for Beginners by Richard Mansfield

- Mapping the Commodore 128 by Ottis R. Cowper

- Inspiration from 'The Last Digital Excess Demo', 'Skeletor Movie',
'Digital
  Acid', C=Hacking - Issue #7, the people on the IRC channel #c-64,
'SYS4096'
  music by AMJ, and Thinktank who tragically passed away months prior to the
  demo but had released his amazing Commodore/Graphics low-res (40 by 25)
  artwork to the public a few years before.

Graphics-Zone is not an overtly arrogant and complicated demonstration.  It
consists of 21 Commodore/Graphics (i.e., low resolution PETSCII) full-screen
drawings (that happen to be *well-drawn*) placed in synch with a very nice
SID tune running in the background.  There is little if any 'Top Code' in
the demo and since I wrote it as a beginner, there should not be anything
exceedingly fancy in the code itself that would preclude its use as a
```

training
vehicle for those not fully skilled in 6502 ML. In fact, there are some half-
baked methods in the code that are rather simplistic even for a beginner.
With this in mind, I hope to share some of the triumphs and pitfalls so as
to instill a motivative spirit in those who seek to learn, and perhaps render
a smile in those who already know :).

As a great deal of you have already seen, running Graphics-Zone is a matter
of loading it and typing "run". The first two screens pop up to form the intro
sequence where two C/G pictures are flipped back and forth slowly to match
the tempo of the music. When the music speeds up, the intro pics are abandoned
for the most part and the rest of the C/G pics in memory fly at you in a mad
attempt to keep up with the tempo. After a minute or two of this, the music
and pics loop back to their start sequences and begin anew. There is nothing
mind shaking about it. The whole demo is basically a planned sequence of
memory moves for the pics with an interrupt (IRQ) driven music player behind
it.

The main block of C/G pics are located from $3000 to $bfff with three
additional pics at $0800-$0fff, $2800-$2fff, and $c800-$cff0. Each C/G pic
occupies 2 kilobytes (KB), the first KB is the screen data while the last
KB represents color memory, as follows:

```
; Memory Structure - C/G pictures
;
; -------------------------------$3000  - 0 kilobyte, start of picture 1
; !   char data  for C/G pic 1
; -------------------------------$3400  - 1 kilobyte, picture 1
; !   color data for C/G pic 1
; -------------------------------$37ff  - 2 kilobytes, picture 1 ends
;
;
; -------------------------------$3800  - 0 kilobyte, start of picture 2
; !   char data  for C/G pic 2
; -------------------------------$3c00  - 1 kilobyte, picture 2
; !   color data for C/G pic 2
; -------------------------------$3fff  - 2 kilobytes, picture 2 ends
```

In actuality, the screen and color data do not extend right up to the next
page boundary (i.e., at the $xxff - $xx00 junctions in hex), they in fact end
at $xxe7 (e.g., $33e7, $37e7, $3be7, $3fe7, etc). For simplicity's sake, I
did not worry about the extra bytes at the end, and I started the each block
of pic data (screen/char or color data) at a page boundary $xx00.

But C/G pics are simply petscii text and color, right? Yes, they do not

come
organized as shown above and for a short time, the prospect of extracting them
into a more usable form seemed daunting. After a few minutes, the solution came to me. I read them off the disk and printed them to the forty column screen on my 128. When this was done, I peeked an image of VIC screen memory
($0400-$07e7) and VIC color memory ($d800-$dbe7) and poked the image to $3000-$33e7 (screen mem) and $3400-$37e7 (color mem). Then I binary-saved them to disk using the monitor (via the 80 column display of course). I repeated this step twenty-one times because I was fortunate enough to have twenty-one quality C/G pics at my disposal :). In retrospect, I later found out that certain "taboo" areas of memory such as the area under the kernel could have been used to store even more data, but at the time I was quite happy to have been able just to switch out Basic and use the space it took up.

What about the rest of the demo, the music and code, eh? Not a problem there.
The music player/data starts at $1000 and heads up to around $26b0 while the main code for the demo itself starts at $c000. I took all the converted pic files (the ones I had binary-saved) and used a packer to relocate them to their final destinations in memory, along with the music and code. Also, because packers conveniently compress all data and code into a run-time executable, I didn't have to worry about providing a front-end ability to be able to "RUN" the ML from basic. The packer took care of all this for me,
all I had to do was specify a start address for the machine-language (ML) code.

The following is a disassembly of the ML code itself. Although I won't go through it 100% step by step, important points shall be perused for the purposes of clarification, and to be honest, for a little self-introspection on my part.

The program starts out quite simply with the start address and a labelling of important memory locations (generally used as pointers throughout the program).

```
;---------------------------------------
         *= $c000; start of program
;---------------------------------------
point    = $fb    ; LSB = point   ($fb)
                  ; MSB = point+1 ($fc)
temp     = $cfff  ; MSB byte storage
temp2    = $cffe  ; used by FLD routine
temp3    = $cffd  ; $cffa/cffb for TBB/AMJ player uses these.
looper   = $ccfc  ; see "pause" subroutine
hit      = $ccf9  ; see "pause" subroutine
;---------------------------------------
```

```
init     lda #$36 ; #$36 is our kill-basic value, so
         sta $01  ; store the kill-basic value into $01 and move the
                  ; basic rom out of the way, exposing the ram underneath.

[AssEd. Note : For general edification, location $01 works as follows in
C64]
[  - bit 0: ROM/RAM at $a000  1=Basic  0=RAM]
[  - bit 1: ROM/RAM at $e000  1=Kernal 0=RAM]
[  - bit 2: ROM or I/O block  1=I/O    0=ROM]
[  - bits 3,4,5 are cassette related........]
[  - bits 6 & 7 are not connected in the C64]
[  - bit 6 checks the status of the caps lock (ascii/cc) key on C128.]

         lda #$00 ; ok...set up accumulator as #$00
         sta $d020; change screen and
         sta $d021; border to black (i.e., #$00)
         sta point; store LSB of pointer (which is now #$00)
         lda #$00 ; initialize the looper
         sta looper

Ok... So what is going on here?  Basically, we are telling Basic to take a
hike so that we can use the memory it once inhabited.  We are also setting
up the LEAST SIGNIFICANT BYTE pointer for the indirect Y function.  This
is a powerful tool in 6502 assembly and we will get to it later :).  The
"looper" is just a memory location that the program will use in its "pause"
subroutine.  Right now, it is set at zero (#$00).


UGLY     lda #$fa ; try and tell tbb/amj music player
         sta $105b; what scanline to play at

amjtune  jsr $1000; jsr call to sys4096/amj's TBB player at $1000


This part is really UGLY.  Basically, I tried to extract the music player
and tried to incorporate it here, but I failed for some unknown reason.  I
decided that since it was proving to be uncooperative that I should just
call
it from here and let it go off on its own.  This presented two problems for
me later.  The first was that the music was playing at raster lines that
were
outside the border area.  In layman's terms this means that it would play
in the middle of the screen and a slight flicker could be seen as the main
code flipped through the pics.  I rectified the situation by finding out
where
it was polling its raster info ($105b) and sticking an #$fa in there.  #$fa
is raster line 250, which should correspond to the lower border on the
screen.
Yes, I could have modified the player code itself, but I was getting a bit
paranoid at this juncture and decided not to modify it.  The second problem
is that by giving up control to the player subroutine, I lost control of
```

```
timing, an annoyance that which we shall discuss later.  Anyways, as you can
see I call the player in the 'amjtune' subroutine and let it do its thing,
but even though it is not in the main code, I've included the player here
for the benefit of the reader.  As this code is from someone else, I cannot
assure a 100% correct disassembly, but read on..


TBB player from SYS4096 tune by AMJ starts at $1000 and proceeds as follows:
(By the way, TBB is AMJ's brother...trivia mode over).

<start of player code>

>        sei        ; disables interrupts
>        lda #$01
>        sta $d01a  ; set up for scan line
>        lda #$7f
>        sta $dc0d  ; enable timer interrupts
>        lda #$35   ; lets play with roms
>        sta $01
>        lda #$00
>        ldx #$00
>        ldy #$00
>        jsr $1100  ; jsr to musix init ?
>        lda #$37   ; let's play with roms
>        sta $01
>        lda #<irq  ; LSB of irq
>        sta $0314  ; stash it
>        lda #>irq
>        sta $0315  ; MSB of irq
>        lda #$3a
>        sta $d012
>        lda #$1b   ; normalize the screen i think
>        sta $d011
>        cli        ; re-enables interrupts
>        rts        ; it used to jmp back to itself, infinite loop
>                   ; as its irq routine played in the backgroud
>
>irq     lda #$01   ; the irq routine
>        sta $d019
>        lda #$35   ; let's play with roms again
>        sta $01
>        dec $d020
>        jsr $1103  ; $1103 (!) I think this jsr's to musix data
>        inc $d020
>        lda #$37   ; let's play with roms yet again
>        sta $01
>        inc selfmod+1 ; self-modifying code!!!
>selfmod lda #$xx   ; #$xx = this is the byte changed by 'inc selfmod+1'
>        and #$01
>        tax
>        lda #$105b,x  ; goes to the scan line table ?
```

```
>        sta $d012  ; sets up the scan line for irq to occur
>        jmp $ea31  ; go to normal c= irq return

<end of player code>

From $105c to $10a0 or so beyond this there is some program data of unknown
function. At around $1100 there is an embedded message by the authors and
then
the music data follows, ending somewhere around $26b0 if I recall correctly.
Please note that this tune is double-speed (i.e., plays twice per frame).

<back to main program code>

The initialization steps have executed and the music player has been told
to start playing.  What is left now is to present the C/G pictures to the
viewer in a meaningful way.

The main1 routine that follows conducts the sequence of the C/G displays.
Its
responsibility is to load and store the MOST SIGNIFICANT BYTE of the address
(location) of each starting pic for a given pattern of displays and then
branch
out to the pattern subroutines, which display c/g pics sequentially given a
predetermined sequence.  Again, the MSB as with the LSB we encountered
earlier
deals with the indirect Y function that we shall explore later.


main1    lda #$30 ; starting pic MSB
         sta temp ; store MSB in temp
         jsr pattern

         lda #$30 ; starting pic MSB
         sta temp ; store MSB in temp
         jsr pattern0

         lda #$c0 ; this MSB is being stored but pattern1 does not use it
         sta temp
         jsr pattern2
         jsr pattern1; fld bounce of the first intro pic only
                     ; located at $3000-$37e7
                     ; fld effect is quite annoying, so its done only once
         lda #$38
         sta temp
         jsr p0      ; p0 routine is a part (subset) of pattern0 routine

         lda #$c0    ; the rest of these are more of the same... calls to
         sta temp    ; pattern subroutines
         jsr pattern2

         lda #$38
```

```
         sta temp
         jsr pattern0

         lda #$c0
         sta temp
         jsr pattern2

         lda #$30
         sta temp
         jsr pattern0

         lda #$c0
         sta temp
         jsr pattern2

         lda #$30
         sta temp
         jsr pattern0

         lda #$c0
         sta temp
         jsr pattern2

         lda #$38
         sta temp
         jsr p0

         lda #$c0
         sta temp
         jsr pattern2

         lda #$38
         sta temp
         jsr p0

         lda #$c0
         sta temp
         jsr pattern2

         jsr pattern3; last pattern before music loops back on itself

         jmp main1   ; by now, music has looped...time to slow down again
                     ; with the original pattern at the start of main1
routine
                     ; (i.e., pattern routine that follows is called at the
                     ; start of main1)


"pattern" is the first of the pattern subroutines.  It's responsibility is
to take care of the first and second intro pics at the start of the program.
It pulls the MSB from the temp memory location (the first intro pic), jsr's
```

to the viewpic routine, and loads a "hit" value which the "pause" routine will then compare with an "looper" value. This determines how long the first
intro pic will be shown. After which, "pattern" will change the MSB value (without affecting the MSB in temp) to point to the second intro pic, and then it repeats this process until the music tempo increases, upon which time
"pattern" gives up control to another "patternX" subroutine.

I mentioned earlier that I lost control of timing when I gave up some control
to the music player code. That is to say, because I could not (at the time) incorporate the player code in here, I had no way of properly synching the pictures to the beat of the music as it played. I corrected this deficiency in true 'newbie' fashion. I used delay loops to form the core basis of what I could approximate as being a "beat" of music. The "pattern" subroutines use
the "pause" subroutine (which includes the core delay loops along with "hit" and "looper" comparison) in order to determine how long a C/G pic should be displayed as the music plays in the background. I had to manually figure out
how long it would take to go from its initial slow tempo to a faster tempo and then calibrate the "pattern" routine to give up control at the transition.

```
pattern  lda temp    ; this sets up the initial -slow- flipping pattern of the
         jsr viewpic ; pics to match the -slow- tempo start to the AMJ tune
         lda #$11
         sta hit
         jsr pause
         lda #$08
         jsr viewpic
         lda #$11
         sta hit
         jsr pause
         lda temp
         jsr viewpic
         lda #$11
         sta hit
         jsr pause
         lda #$08
         jsr viewpic
         lda #$11
         sta hit
         jsr pause
         rts
```

The other "patternX" routines and their subsets (p0, p10, etc.) basically

```
perform addition or subtraction operations on the MSB they initially pull
from the temp location.  By doing so, you can display a number of different
pictures in sequence with a minimum of effort.  Recall that the pics are
saved into memory initially with some organization behind it. That pre-
planning
combined with addition (adc) or subtraction (sbc) operations allowed me to
display pictures in the order I desired.

For example, assume that picture 1 is entitled "Boy", picture 2 is entitled
"meets", and picture 3 is entitled "Girl".  "Boy" is at $3000, "meets" is
at $3800, and "Girl" is at $4000.  The MSB represents the first 2 digits
of the hex addresses I have just given, so "Boy" MSB is $30, "meets" is $38,
and "Girl" is $40.  Notice that each of these MSB's is precisely #$08 hex
numbers apart!  Since we haven't gone into the indirect Y function yet this
may be a little premature, but it is logical to assume that if we had an
indexing system in place all we would have to do in order to move from
picture to picture would be to either add or subtract #$08!  So basically
our pictures are 8 units apart, for simplification as follows:

lda 30       : Our house is at 30 main street :)
jsr viewpic  : Ask a photographer to photograph and display our house
adc  8       : Inform the photographer that the next house is 8 blocks down
jsr viewpic  : Ask the photographer to photograph and display -that- house

In actual code, if I wanted the pictures to come out sequentially as
Boy meets Girl (remember the addresses we specified above) it would be

lda #$30     : tell viewpic that "Boy" is at $3000  (#$30 = MSB = first two
             : digits of the hex number).
jsr viewpic  : display "Boy"
clc          : CLC - Clears the Carry Flag.. required step before addition
adc #$08     : add another #$08 to the "Boy" address..#$30 + 08 = #$38
             : in other words, the address for "meets" which is $3800.
jsr viewpic  : display "meets"
clc          : required
adc #$08     : add another #$08 to the "meets" address..#$38 + 08 = #$40
             : in other words, the address for "Girl" which is $4000.
             : REMEMBER, we are adding in HEXADECIMAL...
jsr viewpic  : display "Girl"

Running this routine (and for now, don't worry about how viewpic works)
within
this program would allow us to display "Boy meets Girl".  Simple, eh?  The
basic concepts hold for patternX routines that use subtraction except that
instead of CLC, you are required to do a SEC before a subtraction operation.

You will notice that the patternX routines jsr to the delay loop (e.g.,
loop1)
routines more directly than the "pattern" routines.  This is because
"pattern"
required a much longer delay and this is why "hit" and "looper" were
```

```
created.


pattern0 lda temp
         jsr viewpic
         jsr loop1

         lda #$08 ; Load oddball MSB
         jsr viewpic
         jsr loop1

p0       lda temp ; Load MSB from temp
         clc
         adc #$08 ; Add #$08 to it
         sta temp ; Store MSB in temp
         jsr viewpic
         jsr loop1
         lda temp ; Bring back MSB
         cmp #$b8 ; Is the MSB at the
         bne p0   ; final pic? $b800
         rts

pattern1 lda #$30
         jsr viewpic
         jsr fldmain
         jsr loop2
         rts

pattern2 lda #$c8 ; Load 2nd oddball MSB
         jsr viewpic
         jsr loop1

p10      lda temp
         sec
         sbc #$08
         sta temp
         cmp #$30
         beq p10
         jsr viewpic
         jsr loop1
         lda temp
         cmp #$28
         bne p10
         rts

pattern3 lda #$c8
         jsr viewpic
         jsr loop1
         lda #$78
         jsr viewpic
         jsr loop1
```

```
         lda #$a0
         jsr viewpic
         jsr loop1
         lda #$40
         jsr viewpic
         jsr loop1
         lda #$38
         jsr viewpic
         jsr loop1
         lda #$30
         jsr viewpic
         jsr loop1
         lda #$08
         jsr viewpic
         jsr loop1
         lda #$60
         jsr viewpic
         jsr loop1
         lda #$a8
         jsr viewpic
         jsr loop1
         lda #$68
         jsr viewpic
         jsr loop1
         lda #$70
         jsr viewpic
         jsr loop1
         lda #$80
         jsr viewpic
         jsr loop1
         lda #$90
         jsr viewpic
         jsr loop1
         lda #$88
         jsr viewpic
         jsr loop1
         jsr loop1
         jsr loop1
         jsr loop1
         jsr loop1
         ldx #$30
         jsr d1
         rts


Ah, here we have reached the famous "viewpic".  You will notice it doesn't
do much.  In fact, all it does is store the MSB pointer for the indirect Y
function and "passes the buck" so to speak to the display routine.  :)

viewpic  sta point+1; Store MSB pointer
         jsr display
```

```
         rts


WARNING : INELEGANT timing solutions up ahead...be afraid, be very afraid...
          ---------

pause    jsr loop1
         inc looper
         lda looper
         cmp hit
         bne pause
         lda #$00
         sta looper
         rts

loop1    ldx #$fd
         jmp d1

loop2    ldx #$01
         jmp d1

d1       ldy #$ff
d2       dey
         bne d2
         dex
         bne d1
         rts
```

"loop1" encompasses "d1", and "d2". The whole scheme is a loop within a loop, with the idea being to be able to get "loop1" to approximately equal one beat of the music playing in the background. After about 30+ (!!!)
recompiles, I got it almost perfect on an NTSC machine. The music plays 17% slower on a PAL (european, australian) machine and so this demo is horribly out of synch in PAL. For those of you who are wondering, Mr. George
Taylor calculated the delay loops to be about 2.8% slower on a PAL machine when taking into consideration the slower PAL CPU and the penalties incurred due to "bad lines" (when the vic chip steals cpu cycles on the bus). As mentioned earlier, the "pause" routine encompasses everything "loop1" has to offer and extends the delay even further. The "loop2" routine would seem to be useless but it is called by the fld-bounce routines. The fld takes longer than a regular display so I figured I would only need to call a delay routine that was a fraction of a music "beat" (which is what "loop1" tries to be).

Now we get to the real nitty gritty of the whole thing. The "display" routine
and its subsets. These make use of the indirect Y function which Mensch and company thankfully chose to implement in the 6502.

```
But wait, we haven't really gone into the indirect Y, have we?  Nope,
because
it is tricky to explain.  Like with movies or sports events, you have to be
there in order to get the feel for it.  In general, it is an index system
that uses a zero page pointer of your choice in order to jump around to
this or that memory location.  But that's too vague...let's really explore
it.

Do you recall that at the start of the code we defined a few labels and
basically equated them as memory locations? We said that "point = $fb" among
other things. That is our memory (zp) pointer for the Least Significant Byte
(LSB) and in the "display" routine below you will see a reference to point+1
which is our memory (zero-page) pointer for the Most Significant Byte (MSB).

An address in memory is made up of two bytes, namely the MSB and LSB.
Think of the MSB as the first two digits and the LSB as the last two digits,
as follows:
                $c000 = $c0 MSB + 00 LSB

                $00c0 = $00 MSB + c0 LSB

The two together make up a 16-bit address and due to how the 6502 works,
it is one reason why 64 kilobytes can be accessed directly (2^16 = 65536).

The indirect Y function allows you to set up an MSB and LSB pointer in zero
page (ie., the first 256 bytes of memory from $0000 to $00ff).  Note : You
can't set a pointer in locations $0000, $0001, and $00ff.  The LSB pointer
I chose was $fb and by definition, the MSB pointer is automatically one
memory
location higher than the LSB pointer, so my MSB pointer is at $fc (point +
1).

The "display" routines make use of the pointer functions as an index to copy
C/G picture data to screen and color memory.  But how does it do it, right?

The "viewpic" routine we encountered earlier sets the MSB to the MSB of the
picture that is about to be displayed.  So for the first intro pic at $3000
this would be an MSB of #$30.  Our LSB was set to zero (#$00) at the start
of
the program.  So we have an MSB of #$30 and an LSB of #$00 = $3000 address.
"display" now sets the Y register to zero (ldy #$00), "pa1" is now ready to
copy memory.

NOTE : For all the examples shown, we will use the MSB of the first intro
pic
       ($3000 = $30 MSB = first two digits).  The rest of the program
changes
       the MSB on the fly so that when the "display" routine is reached, new
       pics can be displayed.  If the MSB didn't change beyond the simple
       incrementations you will see below, we'd be stuck with displaying
       one pic.
```

```
display  ldy #$00  ; "display" encompasses all the memory moves that follow

;--------screen data moves--------------

pa1      lda (point),y ; $x000  $x800
         sta $0400,y
         iny
         bne pa1
```

As you can see "pa1" takes the LSB of the pointer and uses the Y register to increment it (with INY) and then uses the same increment to store values to screen memory (which starts at $0400). In long form, this routine is just doing this :

```
        LDA $3000  ; take the first byte from the stored pic in memory
        STA $0400  ; put the first byte to the first screen memory location

        LDA $3001  ; take the second byte from the stored pic in memory
        STA $0401  ; put the second byte to the second screen memory
location

        LDA $3002  ; take the third byte from the stored pic in memory
        STA $0402  ; put the third byte to the third screen memory location
```

The INY instruction can only increment 256 times before the LSB runs out and it starts looping back to zero, so what we do now is increment the MSB ! (Remember, the BNE instruction checks when Y is equal to zero and moves on to the next routine -- Also, remember that Y itself does not change the LSB pointer, it only adds to the LSB *value* during the loop. LSB pointer itself stays the same, which means we can use it in the next routine without resetting it - the one we do increment is the MSB - via the INC instruction).

```
         ldy #$00
         inc point+1
```

The previous routine "pa1" filled the first 256 bytes of screen memory with the first 256 bytes of our stored pic (i.e., it copied memory locations from $3000-$30ff to $0400-$04ff). We have just increased our MSB by one (using the INC point+1 - remember point+1 = our MSB in zero page). "pa2" will now do
the same thing as "pa1" except it is now working on the next 256 bytes (i.e.,
it will copy memory locations from $3100-$31ff to $0500-$05ff).

```
pa2      lda (point),y ; $x100  $x900
         sta $0500,y
         iny
         bne pa2

         ldy #$00
```

```
         inc point+1

The MSB pointer is incremented again...  $3200-$32ff to $0600-06ff.
Please remember we are using the MSB of the first pic in this example.

pa3      lda (point),y ; $x200  $xa00
         sta $0600,y
         iny
         bne pa3

         ldy #$00
         inc point+1

The MSB pointer is incremented again...  $3300-$33ff to $0700-$07ff ($07e7)
Please remember we are using the MSB of the first pic in this example.

pa4      lda (point),y ; $x300  $xb00
         sta $0700,y
         iny
         bne pa4

;--------color data moves----------------

         ldy #$00
         inc point+1

Are we seeing a pattern here?  :)  The MSB keeps getting incremented so as
to allow yet another 256 byte copy-fill to occur.  But now things change a
little since we will now copy color memory.  Not a problem because when we
first organized our pictures in memory we put color data "RIGHT BEHIND" the
screen data!!  Recall, screen data takes up the first 1 KB while color data
takes up the last KB  (2 KB total per pic).  So what do we do?  We keep
incrementing the MSB until the end of the pic is reached, but now we
redirect
our copy to the VIC color memory area ($d800-$dbff).

"cpa1" copies $3400-$34ff to $d800-$d8ff
Please remember we are using the MSB of the first pic in this example.

cpa1     lda (point),y ; $x400  $xc00 ; we are now in the second kilobyte
         sta $d800,y   ; of our stored pic data in memory.. in other words
         iny           ; this continual MSB incrementation has gone through
         bne cpa1      ; the first KB and has now hit the second KB where
                       ; color memory for the pictures resides

         ldy #$00
         inc point+1   ; increment that MSB yet again..
cpa2     lda (point),y ; $x500  $xd00
         sta $d900,y   ; copies $3500-35ff to $d900-$d9ff
         iny
         bne cpa2
```

```
         ldy #$00
         inc point+1   ; increment that MSB yet again..
cpa3     lda (point),y ; $x600  $xe00
         sta $da00,y   ; copies $3600-36ff to $da00-$daff
         iny
         bne cpa3

         ldy #$00
         inc point+1   ; increment that MSB for the last time!
cpa4     lda (point),y ; $x700  $xf00
         sta $db00,y   ; copies $3700-$37ff to $db00-$dbff ($dbe7)
         iny
         bne cpa4
         rts           ; return back to original calling routine, whatever
                       ; that may be!

;---------------------------------------

nullpic  ldy #$00 ; this is useless, i never did anything with nullpic
                  ; i wanted to expand this to build some kind of random
                  ; pic or blank screen.. but i forgot about it..

;--------fld routine--------------------
```

FLD is known as Flexible Line Distancing and the routine that follows is an amalgam of something The Phantom/FOE did in a Coder's World 3 article. I will
refer the reader to that article and a more comprehensive analysis of FLD in C= Hacking Issue #7. In brief summary, FLD is basically a raster trick that bounces the whole screen up and down quickly without having to engage in moving screen memory or vertically scrolling the actual picture data. This illustrates a good point about coding on the C-64. The best "coders" may not necessarily be the ones who best know the 6510 CPU. Generally, at least in the 'demo world', talent is assessed on how well the individual knows how to properly abuse the ancillary chips VIC, SID, CIA, etc.

The FLD technique from my standpoint was an exercise in trial and error. I sadly did not use a sine table to coordinate the bounce-effect. I basically sat there tweaking it left and right until it did what I wanted it to do on my NTSC machine. In other words, once I got it to bounce the first intro pic a few times and gracefully depart, I was content. On PAL machines the effect is quite skewed and is not recommended for young viewers in the audience as it is rather grotesque :).

```
fldmain  lda #$00
         sta temp2

fld      lda #$2a
         cmp $d012
         bne fld
```

```
         lda temp2
         cmp #$4f
         beq fldmain2

start    ldx temp2
bounce   ldy $d012
         cpy $d012
         dey
         tya
         and #$07
         ora #$10
         sei
         sta $d011
         cli
         dex
         bne bounce
         ldy #$15
         sty $d018
         lda temp2
         clc
         adc #$07
         sta temp2
         jmp fld

fldmain2 lda #$1b  ; recovers first scn
         sta $d011 ; row from fld trick
         clc
         rts
```

This is it! The end of this article and the end of what I hope was an enjoyable experience for you. Part of the disC=overy project is to stave off entropy for as long as possible. The best way to do this is to convert more order out of chaos in our local domain - the world of 64 :). In effect, by increasing the interest and drive we put into these old tanks, we shift entropy and chaos to the rest of the computer world. Because the universe is a closed system, this is the best we can do and is perhaps a lost cause ultimately, but I'll wager no one thought we would get this far.

XmikeX

/S05::$d400:::
::

A Beginner's Guide to the JCH Editor V2.53, NewPlayer V14.G0
by Sean M. Pappalardo (Pegasus/RPG)

```
    The JCH (Jens-Christian Huus) Editor for the C-64
is probably the most advanced sound creator and music
tracking program of the group of software I like to call
"Next Generation" software.  This is software that pushes
the Commodore past the limits the original designers placed
on the hardware. This program has two main functions: to
create and play high-definition analog synthesized sounds,
and to arrange these sounds in music tracks. The program
uses the following features of the SID (Sound Interface
Device - the "Sound Blaster" of the Commodore): Three
simultaneous synthesized voices (viewed as tracks by the
editor), the four basic analog waveforms the SID produces
(triangle, sawtooth, pulse, and noise, hereafter referred to
as "simple sounds"), ring modulation, and filters.

    The secret to the -editor's- ability to produce quality
high-definition sounds is the rapid cycling of simple sounds
through the SID, one right after another, so fast that your
ears more or less fuse the sounds into one complex sound.
For instance, a standard snare drum sound is made up of 5 or
more simple sounds that get played so rapidly that you hear
one sound. The whole process takes only .0026 seconds, which
is how this technique can be accomplished. For reference, a
digital sound is in essence the same idea, except you hear
about 12,000 simple sounds per second as opposed to 360. The
only problem is that digital sound uses vast amounts of
memory (12K per second of sound), whereas a 3 or 4 minute
JCH song takes only 4K, give or take a K or two. Clearly,
JCH songs are the more economical choice.

    The -player- is designed to be called from another program
exactly once per screen refresh. The standard NTSC screen
redraws itself 60 times per second, and so the player
expects to be called 60 times per second, with equal time
between the calls.

    The number of simple sounds that get played per second
can be doubled through what is known as a "double-speed"
player. This simply means that the player gets called twice
per screen refresh, or 120 times per second, so playing
about 720 simple sounds per second. This is done to create
higher definition sounds (like raising the sampling rate of
a digitizer.) The drawback is that twice as much processor
time per screen refresh is taken to play the tune as with
the single-speed player. This technique has been applied in
creating 3, 4, 5, 6, even 8-speed players, each one having a
difference in sound quality, and a proportional increase in
rastertime use. Mind you, the quality is only as good as the
creator of the sound, as the sounds have to be constructed
by the composer, and the faster the player, the more detailed
the sounds must be to sound their best. It is a shame that
```

we can't just use a digitizer at a very low sampling rate to capture sounds instead of having to engineer each one carefully!

The Main Menu

F1 - Load Tables or Music
F2 - Save Tables
F3 - Save Entire Tune
F5 - Send Disk Command
F7 - View Disk Directory
F8 - Enter the Editor

The F1 key on this menu allows you to load up one of three things: an unpacked tune you wish to continue working on (or to view), a set of tables that contain instrument definitions, or another NewPlayer (A routine that actually plays the music. Different versions have different features and drawbacks. For example, 14.G0 uses more rastertime than the 19.G0 player, and they sound close in quality. If I'm not mistaken, the tune in 19.G0 sounded better!)

The F2 key allows you to save the current instrument definition tables from the computer to the disk drive, for use in other tunes, so you don't have to re-engineer all of the sounds again for each tune.

The F3 key simply saves all of the instrument definitions and music data in one unpacked file for later editing or packing (to be used in a demo or otherwise.) My particular version of the editor saves 147-block files, but I understand other versions save 64-block files. The size you get should match one of these.

The F5 key will allow you to send any valid DOS command to your disk drive. (Such as a Change Partition command for CMD device users.)

The F7 key obviously displays a directory of the current disk/partition.

The F8 key is used to go into the actual editing area.

The Editor Itself

Once you make the bold move into the editor, you will see a screen covered with lines and numbers. This is the editing area, and once you understand it, you will get to like it. (I was a beginner too, you know!)

```
    To start, the windows with the light grey "8000" and
green dashes are where the sequence of notes in a particular
tune is displayed. Under that, there are two more windows
with sets of zeroes in them. (These are actually hex bytes
and each has a specific purpose, depending on the specific
NewPlayer in use. For the purposes of this documentation,
I will assume you are using NewPlayer V14.G0 because that
one seems to be most popular.) Each of the digits can be set
from 0-9 and A-F.

To describe all of the keys used in the editor:

  V - Toggles whether or not the cursor moves when editing any
hex bytes. (Useful for testing different values while
playing a tune.) This value is displayed at the bottom right
of the screen. The C+ means that the cursor will move, and
C0 means that it won't.

<Commodore Key> - Toggles keyboard lock. When it's enabled,
only the cursor keys can be used. This is useful for testing
instruments out without screwing up anything you've entered.
It's also good to keep your kid brother from destroying your
tune while you're in the bathroom. When the lock is on, you
will see two white stars at the top left of the screen.

\ - <Used while the cursor is in a track> Sets a marker in
the three tracks at the block which the cursor is in.

F1 - Begin playing tune from the marker set with the \ key.

F2 - Use the computer like a piano, with the current
instrument. (Cursor to the line of the instrument you want.)
This is obviously useful for testing instrument sounds.

F3 - Begin playing the tune at the beginning.

F4 - Stop all sound.

F5 - Switch between block arrangement and block edit.

F6 - Delete a note in the current block, or delete a block.

<INS> - Insert a note in a block, or insert a block when
editing block arrangement.

F7 - Increase octave.

F8 - Decrease octave.

<SHIFT> D - Increase speed of tune.
```

```
<SHIFT> S - Decrease speed of tune.

The speed value is displayed above the cursor toggle
display. (Bottom right) It can range from 0 to 9, so you'll
see S0 if the speed is 0, S9 if it's 9, etc. 0 is the
fastest speed, and 9 is the slowest. The octave is displayed
above the speed, and can range from 0 to 7. Hence you'll see
O0 if the octave is 0, the lowest, O7 if it's 7, the
highest, etc.

Z - Toggle to the instrument parameter window.

X - Toggle to the glide parameter window.

-After using slash (/) to pop up the wave/pulse/filter
window, L, colon (:), and semicolon (;) will bring you to
the waveform, pulse, and filter windows, respectively.

= - Go to the top of the tune. <CLR> will bring you to the
top and put the cursor in the dashes, if it's not already
there.

<SHIFT> F - Fine tuning. This feature allows you to make
miniscule adjustments to the pitch of each track, so as to
prevent/create sound wave interference between tracks. The
values that appear can be set from $00 to $FF. Press
<RETURN> after each one to set it.

<SHIFT> C - Clear all. This will prompt you first, and upon
a positive response, will erase the block sequences, and set
all the blocks to whatever was in the first block (# 00.)

<SHIFT> X - Leave the editor and return to the main menu.
(So you can save.)

<SHIFT> <RETURN> - While in a block, this will insert 16
($10) blank spaces in that block at the cursor position.
WARNING: Don't make any block longer than 5 screens (80
lines). If you do, it will corrupt when packed. NEVER make a
block 8 screens long! (128 lines) This scrambles data from
other blocks in RAM and destroys the tune. If you save it,
you'll never be able to pack it, as the packer will crash.
Even if you delete the long block, the data remains
scrambled and the packer will still crash. You'll have to
rewrite the tune from scratch if you do this. I recommend a
4-screen limit on blocks. (64 lines)

<SHIFT> A - Copy block sequence into buffer.

<SHIFT> Z - Copy buffer into block arrangement.
```

```
  <SHIFT> . - Copy current block into buffer.

  <SHIFT> I - Copy buffer into the current block.

  <CONTROL> 1/2/3 - Toggle voice 1,2,3 on/off.

  <back arrow> - Fast forward (while a tune is playing. Hold
  it down.)

      The left window, which is longer than the other (and
  contains 8 bytes) is the instrument pointer window. The byte
  before the colon is the instrument number. Each row is a set
  of data for that particular instrument. The editor supports
  up to 32 ($1F) instruments. The first nybble in each row
  (the first digit of the first byte) is the Attack control.
  (Attack is how long it takes a sound to reach maximum
  volume.) The second digit is the Decay control. (Decay is
  how long it takes for the sound to go from maximum volume to
  the sustain volume, if one is set.) The first digit of the
  second byte controls the Sustain level. (The volume at which
  the note is held until it is released.) The second digit
  controls the Release of the sound. (Release is how long it
  takes the note to fade to silence after it has ceased
  playing.) A value of 9 or higher in this nybble will create
  an echo effect. All four of these nybbles will be fastest if
  they are set to 0, and slowest if set to F. (This also
  applies to the vibrato bytes, which will be discussed next.)

To recap:
          00:00 00 00 00 00 00 00 00
             ^^ ^^
             AS DR

      The next byte is the Vibrato Speed byte. The first
  nybble controls how quickly the vibrato cycles up and down.
  The second controls how long the note must be sustained
  before the vibrato kicks in.

      The following byte is the Vibrato Control byte. The
  first nybble controls how much the note is oscillated from
  its original pitch, with 0 being none and F being the most.
  The second nybble's function is still unclear to me, and
  changing the value has small but strange effects on the
  behavior of the player.

Review:
          00:00 00 00 00 00 00 00 00
                   ^^ ^^
       Vibrato Speed   Vibrato Control

      The next byte is the 'effects' byte. The first nybble
```

```
  has some applications that are unclear to me, but setting it
  to a '1' will keep the note played a constant pitch, no matter
  what note the track data says to play. (This is useful for
  drums as they have a constant frequency, except for tom-toms.)
  The second nybble controls how the filter information is to be
  applied to the sound. It chooses the high-pass, low-pass,
  band-pass, etc. filters.) A value of '0' in this nybble will
  shut off the filter for that instrument.

      The next three bytes point to filter parameters, pulse
  parameters, and the waveform sequence for this instrument,
  respectively. These bytes are set depending on where in the
  waveform, pulse, or filter tables you wish the sound to get
  its data from. (Explained in a moment.)

Review:                         plse
                                 ^^
          00:00 00 00 00 00 00 00 00
                         //  ^^    ^^
                        fx  filter  waveform

      Now, press the slash (/) key. You will now see a new
  trio of windows, the leftmost of which is the waveform
  sequence. The bytes preceding the colons are reference
  numbers, which are used in the afore mentioned waveform
  pointer byte. The same idea applies to the middle window
  which is the pulse parameters, and the right window, which
  is for the filter parameters.

      The 2-byte waveform sequence lines each represent a
  simple sound. It is in this sequence where all the magic of
  detailed synthesized sound occurs. The first byte controls
  how much the pitch of that particular sound is offset from
  the actual note listed in the track data (That is, if the
  first nybble of the effects byte is not set to 1, otherwise
  these sounds are offset from a constant note.) A value of 00
  makes the pitch exactly equal to the listed note, and as the
  number increases, so does the pitch. (You can't offset the
  note downwards.)

      The first nybble of the second byte selects the waveform
  of this particular simple sound. Values are as follows:
      1 = Triangle
      2 = Sawtooth
      4 = Pulse
      8 = White Noise
  Other values would create interesting sounds.

      The second nybble controls the ring modulation and sound
  nature. For instance, a value of 0 in this nybble will cause
  the simple sound to take the resounding volume from the
```

```
  previous sound and use that to make the current sound, as
  opposed to putting a 1 in this nybble, which will cause the
  sound to generate its own volume, and "stand alone", if you
  will. Other values control the usage and nature of ring
  modulation.

             offset
Review:      ^^
          00:00-00  00:00 00 00 00  00:00 00 00 00
         //     ^^
      ref#      ^fx
          waveform

      Now, to explain the pulse byte settings (Second window
  from the left in the pop-up set. By the way, you can use the
  slash key (/) to toggle these windows on or off, if you
  decide you wish to see more of the tracks.) The leftmost
  byte is the maximum value that the pulse will reach or cycle
  to, depending on the counting direction nybble. The nybbles
  are switched in this byte (as well as in the rightmost byte,
  which is the width at which the pulse wave will start
  cycling from.) So if you place E3 into either of these
  bytes, the pulse width represented is $03E0. The second byte
  from the left controls the speed of the wave. Often,
  interesting waves can be constructed if you set this to a
  high value, resulting in some wave interference, like
  out-of-phase standing waves. Finally, the first nybble of
  the third byte is the counting direction nybble. Different
  values will cause the pulse value to stay steady at the
  start value, go up to the max value, go up then back down,
  infinitely oscillate, etc. The second nybble controls the
  amplitude of the sound wave. Remember, you must have some of
  the waveform data contain a 4 in the first nybble of the
  second byte, because that 4 tells the player to use the
  pulse values specified by the pointer in the instrument data
  window (seventh byte.) See how all this ties together?

      The filter byte settings are exactly the same as those
  for the pulse, except they control the filter (obviously.)
  Remember that in order to use a filter, you must have placed
  some number (other than 0) in the second digit of the fifth
  byte in the instrument data window, and have set the pointer
  (sixth byte) to point to the filter parameters you wish to
  use for that particular instrument.

Review:          speed  amplitude
                    ^^  ^
    00:00-00  00:00 00 00 00  00:00 00 00 00
              //  ^^    ^  ^^ <same for these>
           ref#  end  dir start
```

```
      Regarding the main track windows, the first three dashes
  are for commands that will be acted upon when the music
  reaches that point. These commands can be one of the
  following: instrument change, note slide, or legato. To
  change the instrument, simply type an 'I' in the first
  position, then type the number of the instrument you wish to
  change to. (The number preceding the colon in the lowest,
  longest window.) To use the note slide (or "glide",) type an
  S in the first position, then the number of the glide
  parameter you wish to use at that point (explained later.)
  The slide will affect only the note at that position. (It
  can be extended to other notes if the legato is used.) To use
  the legato, simply type a star (*) in the first position,
  and move off of the line. (The star will change into three.)
  Place this in front of each note that you wish to continue
  from the previous one. The second set of dashes is where the
  actual notes are placed. All you have to do is cursor to the
  desired position, and use the top two rows of the keyboard
  as a piano keyboard in entering the notes, as follows:

 2 3   5 6 7   9 0   - \ <HME>      - black piano keys
Q W E R T Y U I O P @ * ^     <DEL> - white piano keys
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^       ^
C D E F G A B C D E F G A       B

  Pressing <SHIFT> and <RETURN> while the cursor is on the
  second set of dashes will put a sustain flag into that note
  position. (Much like a sustain pedal. It is represented by
  three plus signs +++.)

      The number after each note is the octave, which can be
  raised using F7, and lowered using F8.

Review:

     8000 --- ---
Inst.#00- I00 C#4 - Note: C sharp (#), octave 4
          --- +++ - Sustain
Slid.#00- S00 +++ - Sustain same note, acted on by slide
          --- +++ - "
          --- --- - Nothing (release of previous note)
          --- D-4 - New note, same instrument
          --- +++ - Sustain
Legato-   *** E-4 - New note, sustained from previous
          --- +++                    (no attack/decay)

      The glide parameter window is the rightmost one on the
  bottom of the screen. (And it can be reached by pressing X.)
  The numbers before the colon are the reference numbers for
  each of the glide definitions. The first byte controls the
  direction and extremity of the slide. The second byte
```

```
  controls the speed. A wide range of slides is possible with
  these -two- bytes, which can each be set from $00 to $FF.
  (That's #$ff * #$ff = 65,536 combinations!)

      Editing the arrangement of blocks is fairly easy. The
  light grey numbers have certain significance: The first two
  digits act as a separate byte controlling the transpose
  value of the block. In other words, that byte will increase
  the key of all of the notes in the block by a half-step for
  each number increased. (Don't decrease the number...it has
  strange undesirable effects on the editor or the sounds.)
  The second pair of digits is the number of the block. This
  can range from $00 to $72. Don't go any further, or you'll
  get strange undesirable effects again. (This is probably
  due to the fact that a certain amount of RAM was sectioned
  off for blocks and if you exceed that, you scramble other
  parts of the tune/program.)

      To end your tune, place a block at the end numbered
  $FF00. (This is usually done automatically by the editor.)
  This number causes the player to repeat the music either
  from the beginning or from the set mark, depending on
  whether you have pressed F1 or F3 to start the tune.

      To create more than one tune in the same player
  (conserves space) all you need to do is make the tunes one
  right after the other, then insert a block numbered $FF00
  between each of them, and save.

      Well, that should be plenty to get you started and on your
  way to making tunes! If you have no desire to create sounds
  yourself, you can take them from other JCH tunes very easily
  by using one of the sound ripper utilities, or using a
  depacker and then saving the sound data tables, provided you
  have the same player that the depacked tune was written for.
  The best thing to do is to set up some basic sound
  definitions, and basically play around with some of the
  values, such as the pulse and filter parameters. Once you
  train your ears to break down complex sounds in our world
  into combinations of the four simple sounds, you will be
  able to recreate those sounds in the JCH editor. In some
  cases, you may desire or need a multiple-speed player. All
  you need to do is pick up a one-block program that alters
  the editor to play in multiple speed. (You may even be able
  to alter it yourself.)

Enjoy!
-----
The following is a uuencoded file of Sean's first JCH tune;
begin 600 NewBeginning
MÀ0@K"ÀÀÀES@P."PR,C4ZES<W-2PP.I<U,S(X,"PP.I<U,S(X,2PP.IDBDP!E
```

```
M"À$ÀF2(<P,#ÀP,#ÀP,À?U<3%Q,D<P!_5Q,7$R1SÀ']7$Q<3)',À?U<2[',À?
MU<D<P,#ÀP,#ÀP,ÀÀHÀ@"À)DBD8'ÀP,#ÀP,#ÀP)K'U<3$RX'ÀFL?5Q,3+@<":
MQ]7$R<B!P)K'(,V!P)K'R('ÀP,#ÀP,#ÀPÀ#;"À,ÀF2*1GL#ÀP,#ÀP,#ÀFM3'
MGL#ÀP,":Q\>>P,#ÀP)K'QY[ÀFLC(GL":QRÀ@S=3(GL#ÀP,#ÀP,#ÀÀ!0)!À"9
M(I&9P,#ÀP,#ÀP,À%Q\K&QLF9PÀ74RL;)F<#À!=3*QLO9F<À%U-G-("#9F<#À
MP,#ÀP,#ÀÀ$\)!0"9(I$>P,#ÀP,#ÀP,"5RL;&R<@>P)74U<3+'L#ÀE=35Q,G9
M'L"5U-D>P)7-(-D>P,#ÀP,#ÀP,ÀÀCÀD&À)DBD9KÀP,#ÀP,#ÀP,#ÀP('(V9KÀ
M@<?'FL#ÀP,"!Q\>:P('(R)KÀ@<?(FL#À@=3(FL#ÀP,#ÀP,#ÀÀ,D)!P"9(I&<
MP,#ÀP,#ÀP,">U<;&R\B<P)['RL;&R9SÀGL?'G,">R,B<P)['R)SÀP)['R)SÀ
MP,#ÀP,#ÀPÀÀ&"@@ÀF2*1'\#ÀP,#ÀP,#À!<K&TL;+'\À%RL;2QLL?PÀ7*RQ_À
M!<K+'\À%RLL?P,À%RLL?P,#ÀP,#ÀP,ÀÀ-ÀH)À)DB("À@("À@("À@("À<4""<
M02"64"">4"À%02!,()Y!()92()Q$(!Q/À'(*"@"9(A$@'%"64D6>4T5.!513
M($].(#4OGC(Q+Y8Y-!PZ(!*7)YA!FRÀ%3D57($)%1TE.3DF;3IA'ER<ÀIÀH+
MÀ)DB("À@("À@("À@("À@("À@("À@("À@$II!GRÀ%1U)%050@5%5.GT6:
M(0#9"@PÀF2(1'BV9/05214Q%05-%1"!/3B!42$4@1$%9($E4(%=!4R!#3TU0
M3$541429/1XMÀ!<+#0"9(A&<1Y9219E%5$D%3D=3(%1/FSH@ETH>19]&F48@
M!4)!F4.?3QY,ET\@*!Q:GTF<4AY#'S";*2PÀ4@L.À)DB("À@("À@("À%4I9!
M0QQ(14P@2$5,1T664T\%3ILL(À5!GTX?1T5,3R!"24%.0Y](!4F;+À"4"P\À
MF2*1("À@("À@("À%5)E/'DT@2T$>39E)!4Z;+"À%19E$GU>:29=.G"!6'T&5
M3BÀ<4X%%EDZ;5)Y%!4Z;+À#*"QÀÀF2)!3D0@04Q,($]&($U9($]42$52($92
M245.1%,@04Y$($-/3E1!0U13(CJ>,SÀW,@#A"Q$ÀGC,P-S4Z@4&R,*0V.H(Z
MB3$WÀÀÀÀ_P#_À/\À_P#_À/\À_P#_À/\À_P#_À/\À_P#_À/],0ÀQ,V@P!À@0/
M0#/K:1@5$0'\/C0PÀÀÀÀÀ/[^_@ÀÀ("U03$%915(@0ED@2D-(+BXN+DU54TE#
M($)9(%E/52$M"@H*J*(ÀN>\3G4X3G503N?À3G5$3G5<3R,CHXÀ/0YZ(ÀCDT3
MN>\3C0L,C7L3C7P3C7T3G=D3[DT3K0L,&'GO$XT+#!AM31/HX!#0YZT@#/ÀK
MH@*Y\!.-31,]6A.=!@S*$/$L31,0%:(ÀN?$3G503N?(3G5<3R,CHXÀ/0[:ÀÀ
MF)DÀU,CÀ&]#XJ)E^$YEF$\CÀ#-#UK0D,C1C48*("O6D3R0+0++QR$[EC%+Q@
M$YD%U+QR$[ED%+Q@$YD&U*U*%/À)K4D4F0343À\-O7@3F034RA#*I?M(I?Q(
MH@*]!@S0ÀTRÀ$KUO$_À(WF\3T!-,O0Z\<A.Y9A0I#]U[$_À&WGL33)H/G6\3
MO743G7L3O4X3A?N]41.%_*ÀÀF)UL$['[$À\*G8$3_DX3TÀ/^41/(L?NHN9<5
MA?NYT16%_+QF$['[,"CP',E^\ÀZ=?A.]RA/P"=[*$TR7#?YL$ZG_G6,3T&RI
M_IUC$_YL$]!B2"G@R8#0&6A(*1"=;!-H*0^HN=D3G7L3G743_F83T+')H-À,
M:ÀH*"IUR$_YF$]"A:"D_"JBYM!2=P1.YLQ1(*1^=Q!-H2"FÀG<<3J0&=RA.I
MÀ)W0$YW3$V@I(-#._LH3T,G^9A.\9A.Q^\E_T"VIÀ)UF$ZB]3A,8:0&=3A.%
M^[U1$VDÀG5$3A?RQ^\G_TÀR]5!.=3A.]5Q.=41/)_MÀ.J0"=!@R\8!.9!-1,
M@!*];!/P([UO$]À;O6,3G1H,O7X3G10,O8$3G1<,O<H3G<T3G6D33)H/O&À3
MK4@4F074F0;4O6\3\"],@!*\8!.]>!,I_ID$U+QR$[EC%+Q@$YD%U+QR$[ED
M%+Q@$YD&U+UX$YD$U$P$#[UC$YT:#+U^$YT4#+V!$YT7#+W*$YW-$YUI$[UL
M$_À#3)H/O&À3K4<4F074F0;4K4H4\ÀNM210I_ID$U$P$#[UX$RG^F034O'(3
MF)T=#+EJ%)V^$[EI%)VK$ZBIÀ)VN$[E5%"FÀR8#P#[E6%$@I\)VQ$V@I#YVT
M$[QR$[EG%*ÀÀ*0_P&,D(\!,*"@H*#0D,C1C4R*T*#!U:$]À'R*T*##U=$XT7
MU(T*#,À!T!Z\<A.Y:!2-NQ.HN4D4*8#)@-À&N4H4C;T3J0"-O!.IÀYUI$ZU*
M%-À#3,<0WFD33(À2O:X3\À;>KA-,\À^\JQ.Y4Q1(2DI*2IV?$V@I#YVB$[E4
M%)VE$[E5%(U-$RD_"IVN$RQ-$QÀ4N5842"GPG;$3:"D/G;03J0"=J!,L31-P
M!Y@8:02=JQ.]J!/0&;VQ$QA]I1.=L1.]M!-IÀ)VT$]VB$]ÀA\!>]L1,X_:43
MG;$3O;03Z0"=M!/=GQ/0"+VH$TD!G:@3XÀ#PÀTS'$*V\$_À&SKP33)H0K+L3
MN4<42"GPC;<3:ÀH*"@J-N!.Y2!2-N1.Y212-31,I/PJ-O!,L31,0)*T*#"D/
MC4T3N4H42"GPC;T3:ÀH*"@H-31.-"@R-%]2IÀ(VZ$RQ-$WÀ'F!AI!(V[$ZVZ
M$]À1K;T3&&VY$XV]$\VX$YÀ9LÀ^MO1,X[;D3C;T3S;<3LÀBMNA-)À8VZ$[QR
M$[EG%"GPR1#0'[R^$[G_$\E_TÀJY(Q2=OA.HN?\3G0\,J0"=#ÀQ,,1&\OA.Y
M_Q,P$,E_T!*Y(Q2=OA.HN?\3$À8*HÀ%,&Q$8?10,"AA]%PR@À(Q-$ZBYC1(8
```

```
M?=83G0P,N8X2:0"=#PR\OA.Y(Q2=>!/^OA.]S1/P4[W'$]À6O=À3&'W!$YW0
M$[W3$WW$$YW3$TQP$;W0$SC]P1.=T!.]TQ/]Q!.=TQ.M31/0$[T,#!A]T!.=
M#ÀR]#PQ]TQ.=#PR]:1/)À?À#WFD33%82O6D3\#[)À?À&WFD33%82O'(3N684
M2DI*2IV'$SCIÀ9V$$ZDÀG9À3G983G9D3N6442"GPG9P3:"D/"IV3$]YI$TQ6
M$KV'$_!YWI,3,À-,5A+^DQ.]#PQ*2DJ-"PR]G!,8;0L,G8H3J0!IÀ)V-$]Z$
M$QÀ.O9À320&=D!.]AQ.=A!.]D!/0%KV6$QA]BA.=EA.]F1-]C1.=F1-,0Q*]
MEA,X_8H3G983O9D3_8T3G9D3O0P,&'V6$YT,#+T/#'V9$YT/#+Q@$[VQ$YD"
MU+VT$YD#U*V]$XT6U+T,#)DÀU+T/#)D!U+UX$ST:#)D$U,HPÀTP:#6B%_&B%
M^VÀ6À2<!.À%+À5\!<P&*À:$!N@'4À?À!#@(MÀDX"<0*6ÀKT"YP(3ÀT(#=À.I
MÀ^À#&P1:!)L$X@0L!7L%S@4G!H4&ZÀ91!\$'-PBT"#<)QÀE7"O4*GÀM.#ÀD-
MTÀVC#H(/;A!H$6X2B!.O%.L5.1><&!,:H1M&'00?W"#0(MPD$"=>*=8K<BXX
M,28T0C>,.@@^N$&@1;A)($Z\4JQ7Y%QP8DQHA&X8=1!\<(-ÀBW"30)QXI5BO
MR+G@Q)C0"-TPZB#X+OTÀN@1/&!D9M@%,&!D9À0($_OW[ÀÀ<._O[^ÀÀL"ÀÀÀÀ
MÀ0$!ÀÀÀÀÀÀÀ@$Q,_04$A#P,'-#ÀÀÀÀÀÀ__\ÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀ
MÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀ&!@À$À1V@X,À"@H(À+À/À90:W@$!%0ÀÀÀÀÀÀ
MÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀ!À@,'"P\3%QL?(R<K+S,W.S\ÀÀÀ#\/À&V%À$5
M3!4#![84À15,%?__À'_?$@[?#'_?"@@&!0!_<!]P?PÀÀ?P!_ÀÀ0'?PÀ%!W\À
MÀ@=_00"!$$"À$À6!$!À0$ÀÀ(@1&À$4$A%$$6(2$A&"$A(1PA(2$@#PÀ)À0IÀ
M@@0ÀÀÀÀÀÀÀ#_"ÀÀÀ_]\ÀÀ@):ÀÀÀÀÀ"4IÀÀÀÀÀÀ@À#N4ÀÀ!ÀÀÀÀ@("ÀÀÀ$ÀÀÀ
MÀ@54ÀÀ(1!À0//_@ÀÀÀÀÀÀ!,/J0ÀÀÀÀÀÀ%@53ÀÀÀÀÀÀÀ8!5,ÀÀÀÀÀÀ!P%4PÀÀ
MÀÀÀÀ(ÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀÀ@À"ÀÀÀ,À!0À'!PD)"PL-"P\/$1$ÀÀÀ,À!0À'!PD)
M"PL-"P\/$1$ÀÀÀ,À!0À'!PD)"PL-"P\/$1$ÀÀÀ,À!0À'!PD)"PL-"P\/$1'_
M@À&ÀÀ00!!@$("ÀH*#ÀP.#!À0$A(!À00!!@$("ÀH*#ÀP.#!À0$A(;'!T>'RÀA
M(B,D)24F)R@H*2D!À00!!@$("ÀH*#ÀP.#!À0$A+_@À*ÀÀ@("À@("À@("À@("
MÀ@("À@(3$Q,3$Q05%146%Q<7&!D9&1H3$Q,3$Q05%146%Q<7&!D9&1HP,3(P
M,SÀT,34Q-C$W-C@Q.3'_"QDI+CM*5V9S@H^>J[K'UN/R_PXF/55LA)NSRM3;
MY/'Z!RÀY4FMXA9*KQ,G.T=37VN#EZ_'W_0,)#Q86%A86%A86%A86%A86%A86
M%A87%Q<7%Q<7%Q<7%Q<7%Q@8&!@8&!@8&!@8&!@8&!@8&!@8&!@9&1F@@#"$
MÀ(ÀWAÀ"À-(0À?Z""À(ÀTAÀ"À,(0À@#>!À'^/À($À?XÀPAÀ"À-X0À@#6$À'^"
MÀ(ÀUAÀ"À,(0À@#>!À'^À,(0À@#>$À(ÀRAÀ!_@@"À,H0À@#"$À(ÀW@0!_@"V$
MÀ(ÀTAÀ"À,H0À?X(À@#*$À(ÀMAÀ"À-($À?XÀMAÀ"À-(0À@#&$À'^"À(ÀQAÀ"À
M+80À@#2!À'^À*80À@#"$À(ÀMAÀ!_@@"À+80À@"F$À(ÀP@0!_@"F$À(ÀPAÀ"À
M*X0À?X(À@"N$À(ÀIAÀ"À,($À?XÀKAÀ"À,H0À@#"$À'^"À(ÀPAÀ"À*X0À@#*!
MÀ'^À*X0À@#*$À(ÀOAÀ!_@@"À+X0À@"N$À(ÀR@0!_I(À8AÀ"À&($À@!B!À**À
M&($ÀI(À3@0!_I(À8AÀ"À&($À@!B!À**À&($À@!B!À'^D@!6$À(À5@0"À%8$À
MHHÀ5@0"D@!"!À'^D@!6$À(À5@0"À%8$ÀHHÀ5@0"À%8$À?Z2À$80À@!&!À(À1
M@0"B@!&!À*2À#($À?Z2À$80À@!&!À(À1@0"B@!&!À(À1@0!_I(À3AÀ"À$X$À
M@!.!À**À$X$ÀI(À.@0!_I(À3AÀ"À$X$À@!.!À**À$X$À@!.!À'^E@#R*À(À^
MAÀ!_A0"À0(HÀ?X!!B@"À0X0À?X!!AÀ"À0(0À@#Z$À'^À0(HÀ@#Z$À'^À0(0À
M@#R$À(ÀWAÀ!_@#F!À(!À@0"À/H$À@$"!À(ÀY@0"À0($À?XÀ^@0"À0($À@#F!
MÀ(!À@0"À/H$À@$"!À'^À.8$À@$"!À(À]@0"À0($À@#F!À(!À@0!_@#V!À(!À
M@0"À.8$À@$"!À(À]@0"À0($À?XÀUB@"À-8$À@#6!À'^À-80À@#>$À(À\AÀ!_
M@#6$À(ÀYAÀ"À/(0À?XÀW@0"À/($À@#Z!À(ÀW@0"À/($À@#Z!À'^À-X$À@#N!
MÀ(À^@0"À-X$À@#N!À(À^@0!_CP"!À'^/À($À?XÀÀ?XÀÀ?XÀÀ?XÀÀ?Z://(%^
M?X]^@7Y_IX\\@7Y_J(\\@7Y_IX\Y@7Y_IH\Y@7Y_IH\U@7Y_J(\U@7Y_IX\W
M@7Y_IH\W@7Y_&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
M&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
9&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
À
end
```

```
/S06::$d400:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::


                      An inside look at MODplay 128

                           +-+     by     +-+

                    Nate Dannenberg and Mike Gordillo


Nate Dannenberg is an expert at digitizing sounds on the Commodore 64 and
128.
For the past year or so, he has been the heart and soul of probably the most
revolutionary program in Commodore 64/128 audio since Mark Dickenson's
Stereo
Sid Player. MODplay is the name and it will rock your system to the core.
You
are the first priviledged readers to get a sneak peek at this exciting bit
of
software.
--

> Nate, you are the author of the world's first and only .mod player on the
> 8 bit Commodores.  I am curious to explore some of its inner workings.
>
> But first... What exactly IS a "mod" anyways?

Mike, you probably know the answer to that one already.  A mod is simply
a music file format which started on the Amiga line of computers and spread
to many other systems.  It is a very efficient format in which music is
sampled
or cut into discrete portions and later reassembled in many different ways
to
create varied tunes and patterns to form a musical score.  The module format
can be summed up as a form of musical archive.  A few instruments may be
sampled only once, but their samples can be manipulated in thousands of ways
that may not be possible on the actual instruments.  For example, a sample
of
a flute can form an entire orchestral piece in and of itself without much
effort.  However, in order to achieve this, a computer requires a moderate
amount of speed and memory.  Many have discounted the possibility of .mod
playback on our machines and have therefore not explored the true limits of
hardware in this regard, until now.



> The C128 is not known for top speed and memory though.

Fortunately, it has just enough speed and memory when combined with a 17xx
```

Ram
Expansion Unit (REU). To be perfectly honest, my modplayer can only function
with an "REU" and a C128 running at 2 MHz. But it functions well enough to keep tempo with the majority (if not all) the 4 track "mods" out there and it can even output sound in 8 bits! In the future I may try to support the Ramlink/Ramdrive units and perhaps one or two of the more popular internal ram (256k to 1 megabyte) expansions. Although the Ramlink/Ramdrive units may not have enough speed for the task, I do believe the CMD Super 64 and 128
accelerators will change this and allow for potentially both C64 and C128 accelerated modplayers in the future.

> But for many years, I have heard (from Amiga users primarily) that our little
> machines did not have the CPU horsepower necessary to simultaneously process
> and play back the module format. Amiga users cannot be wrong! How did you
> manage to do it?

You have to know a little about how the module format works, but without going
into too much detail, imagine all samples within a .mod file as being organized
into rows and patterns. These rows and patterns determine what, when, and how
sample data gets played back to the user. In my player, rows and patterns are
centered around their respective tracks (channels). The basic procedure is described as follows :

   Get the 16 byte row data from the ram expansion unit. Now divide the row into four 4-byte subsections where each subsection represents a track (1,2,3,4). Get the current 1 KB pattern data into memory and extract the next 16 byte stub from that pattern. Divide the pattern stub as the row was divided before (into four 4-byte subsections) to match each track.

Within each four byte track, pattern information about a row is divided into a sample number, a note period, an effect command, and an effect parameter. The sample number directs the program to use a certain sample for whatever row
(i.e., notes) are being played in that track. The sample number is derived as follows : "zeroth" high nybble byte + (second high nybble byte/16). It is
then used to index a data table where info such as sample length and start address inside the ram expansion unit are obtained. The note period, on the other hand, tells the CPU how many 3.5 MHz ticks to count down before sending
out the next sample byte. The note period is then used as a lookup into a

frequency stepper table in order to determine the pitch of the note. A value
of 254 for example would be the equivalent of concert pitch A-4. The effect command modifies the pitch, playback time of the note or the whole pattern, and the volume if needed. You can use it to apply an array of effects to each note, including vibrato, portamento, arpeggio, etc. The effect parameter
in turn modifies the effect command. For example, if the effect command is a zero and its parameter is a zero as well, then any effect for that track will be cancelled immediately. Likewise, if the effect command is zero and its parameter is something other than zero, then the effect command is treated
as an arppegio using the supplied parameter value. For reference, if any pattern value for a given row is zero then the pattern data for the previous row is used.

After all is said and done, my program grabs all the module data in little spurts from the ram expansion unit at approximately 459,200 bytes per second leading to (after processing) an effective 8.2 KHz play-back rate. It is interesting to note that my modplayer could reach nearly twice the sample rate it has now if it were to pull its data from the 128's internal memory. Alas, there are few .mod files that could comfortably fit in 128 kilobytes. If I decide to allow the player to grab data from internal memory, it would probably be in support only for internal ram expansions.

> How would internal ram be faster as compared to the ram expansion unit?

The ram expansion unit (REU) is a marvelous blessing. It allows my program to naturally store huge .mod files and has the fastest transfer rates of any ram storage device in the Commodore 8 bit market. Despite this, it does carry overhead penalties that simply do not exist when accesing internal memory. For example, I must calculate a three-byte value into the ram expansion unit (LSB, MSB, RamBank) and copy it to the ram expansion address pointer and also into internal memory. I have to toggle down to 1 MHz while accesing the REU. I then toggle back up to 2 MHz while copying the module data into the C128 as I do a 4-level deep interleave on the module data itself. As this entire sequence is repeated many many times, you can see how much time is lost. If I were doing all this from internal memory, I would have to again calculate the three-byte pointer but this time I could keep it in zero page and would never have to copy it anywhere (except to copy the bank pointer to the memory mangement unit inside the C128). I would never have to switch to 1 MHz mode. Furthermore, I would not need to access the module data in small chunks as it would all be easily available in internal memory. As it stands now, data is cached into the C128 as described earlier for logistical reasons concerning ram expansion overhead times. A more direct method of accessing the module data would be simple with internal memory.

> You mentioned that your modplayer outputs in 8 bits. Could you explain

> how you squeezed 8 bit audio out of the C128.

Originally, I used a technique called Pulse Width Modulation to do it. Remember that early 64/128 digiplayers used the $d418 reg. for 4 bit digis. I used $d402 instead as the DAC register while toggling the test bit in $d404
for each sample byte. I then filtered voice 1 and WHAM, instant 8-bits. The
techinique is interesting and might be of use to you, as follows :

Simply put, just set voice 1 to produce a pulse waveform for the current sample rate. This pulse will range from zero to maximum width. At the sample rate of my player, max. width corresponds to 127 cycles. Now start stuffing numbers into $d402. Each time you do that, put a hexadecimal 49 and then 41 into $d404 and be sure to filter voice 1 with a low pass..voila. Your timing needs to be good and steady. Needless to say, the whole operation
works best in 2 MHz mode.

Using pulse width modulation, the pulses can range from 1 cycle to about 127 cycles wide at the 8.2 KHz sample rate. That is equivalent to 7 bits and the
filter gives you a boost in quality (i.e., effective 8 bit resolution).

The one major drawback with the PWM technique is that it leaves behind a residual squeal that is audible at the current sample rate. I am currently working on a methodology to completely eliminate the squeal and the most promising option appears to be an extension of the PWM technique known as PCM or Pulse Code Modulation. There is also a minor problem with low volume output but this is easily corrected by amplification at the user's sound system.

> These 8 bit techniques you mentioned are for use with the SID chip. Do you
> have any other ideas in mind?

If the SID had a volume register from 0 to 255, I could have used the standard
$d418 DAC technique for 8 bit resolution. As this is not the case, PWM/PCM is necessary for the quality I hope to get out of the SID without resorting to additional hardware. I do provide standard 4 bit and dithered 4 bit sound
output in the program and will fully implement 8 bit SID sound when I feel it
has met my requirements. However, I have not discounted other options. As seen elsewhere in this first issue of disC=overy, I have gone into detail on an 8 bit DAC board that is very simple to build and program. The modplayer already supports this particular board as a low cost alternative. I am also working on a more advanced dual DAC -- ADC board that will be available for sale along with the modplayer. It will have four tracks, each of 8 bits

```
and with an 8 bit linear volume control yielding an effective 16 bits of
resolution per channel :).


> Before we depart, I would like you to emphasize the difficulty in what you
> have done, a C= 6510/8502 modplayer!  :)

The task was not terribly difficult in theory.  In practice it has sometimes
become quite a hassle and it is still in progress.  If you want a
comparison,
for a no-frills .raw digi format (not the module format) you could probably
get away with over 80 (eighty) KHz in simple 4 bit mono on a C128 at 2 Mhz
and you can do about 60KHz 4 bit mono from the ram expansion unit.  On a C64
you should be able to do at least 30 KHz mono 4 bits from the ram expansion
and about 44 KHz from memory.  Obviously, these would be short samples but
the
drive home point is that the module format is indeed a computation-intensive
exercise when compared to .raw digis.  Regardless, my player can handle
up to 31 samples, up to 255 patterns, 4 to 8 bits, 4 tracks, and up to
two megabytes of .mod file if you have the 2 meg REU expansion.  I have not
yet implemented the sliding routines yet, and its sample rate of 8.2 KHz,
although not exceedingly high, is quite adequate for many .mod files.  I do
believe that the new SuperCPU 64 and 128 accelerators from CMD will allow
for even greater sample rates and generally make my task easier overall.  I
will take a close look at them and at internal memory expansions.  If I find
these "vehicles" to be proper and good, I may support them in the modplayer.

> I'd like to thank you for granting this interview, Nate.

Well, just don't let it go to your head, Mike.  :)

> ;)

--
For more information on this one of a kind product, you may reach Nate at
the following addresses:

Dannenberg Concepts, Ltd.
"Bringing your Commodore 128 one step closer!"

Email:   tron@wichita.fn.net
Phone:   (316) 777-0248
Snail:   Dannenberg Concepts, Ltd.
         c/o Mr. Nate Dannenberg
         306 S. 1st Ave
         Mulvane, KS   67110
FidoNet: 1:291/25
         Groups:  CBM, CBM-128, and CBM-GEOS
Usenet:  comp.sys.cbm, alt.binaries.sounds.mods
```

```
/S07::$d600::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::

Some preliminary data on VDC timing by SLJ 1995/96 (comments ->
sjudd@nwu.edu)
-----------------------------------
                                    With some very helpful commentary from
                                    Andreas Boose!

        The 80-column VDC chip on the C128 is one of the least documented
features of the 128.  All documentation I have found echoes the 128 PRG,
so that the general features and programming of the chip is well known,
but as far as I can tell there is zero information on timing, internal
operation of the VDC, etc.  In an attempt to gain some insight into
the timing and operation of the VDC I wrote a series of simple experiments.
Later I received some helpful suggestions and comments from Andreas Boose:
any comments indented with a '>' are from him.

        My personal interest is in using the VDC as a graphics coprocessor.
As anyone who has programmed the VDC knows, all communication with the
VDC is done through two memory locations, $D600 and $D601, and entails
the use of a busy-loop to determine when the VDC is ready to be read or
written to.  My idea was to have the 8502 doing something useful instead
of sitting around waiting for the VDC, so that I could for instance do some
calculations while the screen is being cleared.

        Attached are some simple programs I wrote to get a feel for VDC
timing.  There is an ML program to do the VDC operations, and a BASIC
program to generate a little statistical analysis of data.  Eight tests
are performed and timed using CIA #2 timer A.  In retrospect some are
pretty naive, but I include them all for the sake of completeness:

        Test 1: This is a simple calibration test to determine how many
                cycles are used in starting/stopping the timer.  It
                simply starts the timer, executes 5 NOP instructions,
                and stops the timer.  By subtracting 10 cycles from the
                time elapsed, you get the overhead.

        Test 2: Load a register (Reg 18) -- This test simply times how
                long it takes to tell the VDC which register you want
                to operate on -- no reads or writes are performed.  The
                idea is to calculate how long it takes the VDC to find
                a register.  Note that the register is not loaded with
                anything -- by "loading a register" I simply mean making
                it read/write accessible.

        Test 3: Load a single register twice.  My idea here was to see
                how 'smart' the VDC was, i.e. if it already had the
                register loaded, would it spend any extra time reloading
```

```
                it.  In principle this should be twice Test 2 above.

        Test 4: Load all registers.  This test starts at register 36 and
                goes down to zero.  The idea here is to see if different
                registers take different amounts of time to fetch, i.e.
                will we just get 37 * Test 2 here?

        Test 5: Two reads of register 18.  The whole point of this test is
                to see if _reading_ from a loaded VDC register affects
                subsequent loads in any way (the expectation is that
                it would not).  In principle this test should be Test 3
                plus an additional 8 cycles for the 2 LDA's.

        Test 6: Read a single VDC memory location ($1919).  This is a test
                to see how long it takes to find and read a memory location
                within the VDC RAM.  Location $1919 was used simply because
                the value $19 was already in the accumulator :).  This one
                should take a little more time than three register accesses.

        Test 7: Loop memory fill.  Using a loop on the 8502 side, this test
                fills 256 bytes of VDC memory, starting at location 0,
                using the auto-increment feature of the VDC.

        Test 8: Block memory fill.  This test uses the block-write feature
                of the VDC to fill 256 bytes of VDC memory (0-255).
Naturally
                the idea is to compare it to Test 7.

        Just to make sure the test 7 was doing what I wanted it to do I
stored the VDC memory location after the loop, to make sure it was $0100
(which it was :).

        The BASIC program runs these tests a bunch of times (parameter N
in the program) and calculates a few averages.  The data is all stored
in arrays so that it can be viewed, sorted, or whatever.

        The first time I ran these experiments I did so in SLOW mode and in
FAST mode.  In slow mode an extra 43 cycles would pop up from time to
time.  As several people informed me,

>This has not anything to do with interrupts, it's the VIC-IIe's DMA which
>halts the 8502 on any bad-line for 40-43 cycles. To avoid this jitter
>switch off the screen before measuring, lda#$0b:sta$d011:lda$d012:bne*-3.
>Note the loop after switching the screen off. It's important to wait until
>the raster counter is out of the text area as switching off the screen
>doesn't affect the current screen. After measuring you can switch on the
>screen with lda#$1b:sta$d011.

These extra cycles disappear in 2MHz mode, however:

>Yes, switching to 2MHz forces the VIC-IIe to leave the phi1/2 DMA cycles
```

```
>(almost) untouched. This is a very brutal way to get rid of the 40-43 DMA
>cycles since the VIC's core doesn't know that you want him to leave these
>cycles untouched until you have switched off the screen. So it still tries
>to perform DMA. As this would mean a bus collision with the 8502 a
>protection circuit inside the VIC shuts down its address lines, AEC and BA
>on DMA times. The VIC never gets the data it wanted to display a picture,
>it gets the data the 8502 has accessed - this gives are real bogus screen.
>:-)

As a comparison, two runs follow; the second run is in SLOW mode, with VIC
turned off, and the first is in FAST mode.  Both runs do 120 repetitions
(N=120):

        FAST mode, N=120:

        Test 1: Overhead = 1.4 (expected=1.5)
        Test 2: Single register = 4.9 (expected=5)
        Test 3: Single reg twice = 10.14166... (expected= 9.8)
        Test 4: total time= 304.166...
                minus loop overhead= 212.166...
                div 37 BIT-loops = 27.166...
                => bit-loop repetions = 7.7619
        Test 5: Two reads REG 18 = 14.36
                Expected = 13.8 (twice Test 2)
        Test 6: One memory fetch = 115.66...
                Expected = 20.7  diff = 94.966...
                Probable waits for VDC Reg 31 fetch = 27.133...
        Test 7: Loop memory fill= 10755.766...
                Extra VDC waits= 8347.866...
                Avg BIT-loop repetitions = 9.3168
        Test 8: Total block fill= 179.94166...
                => Approx VDC time spent on 256 byte block fill=
153.24166...

        I include the above just for the sake of comparison; my 'expected'
values are very naive, and the numbers themselves are not terribly
insightful.

        SLOW mode, VIC disabled, N=120:

        Test 1: Overhead = 3 (expected=3)
        Test 2: Single register = 10 (expected=10)
        Test 3: Single reg twice = 20
        Test 4: total time= 554
                minus loop overhead= 370
                div 37 BIT-loops = 0
                => bit-loop repetions = 0
        Test 5: Two reads REG 18 = 28
                Expected = 28 (twice Test 2)
        Test 6: One memory fetch = 118.766...
                Expected = 42  diff = 76.766...
```

```
                Probable waits for VDC Reg 31 fetch = 10.966...
        Test 7: Loop memory fill= 11614.5333
                Extra VDC waits= 6747.533...
                Avg BIT-loop repetitions = 3.76536458
        Test 8: Total block fill= 197
                => Approx VDC time spent on 256 byte block fill= 143

Comments: Tests six and seven are the only tests which showed variation;
all other tests stay constant throughout all trials, with one notable
exception:

Test #8 weirdness: _Very_ occasionally a spurious number will come up,
and always only on the first trial.  It is possible to duplicate:
from the BASIC program, set N=5 say (so you don't have to wait
forever).  When the first tests are reported and the program pauses
for input, hit run/stop -- the cursor should now be red.  Then rerun
the program, and watch the last number printed.  It will usually be
200, but sometimes numbers such as 1019, 1012, 592, and 963 pop up on
the first trial.  If you break the program at other places, though,
these numbers don't come up.  This is a mystery to me.

        1MHz mode with VIC turned off is the most reliable indicator of
VDC's performance:

        Test 3 is indeed just twice Test 2, which is not surprising for
register 18, since the BIT-loop is never executed.  (Twice Test 2?  Timid
termites tremble, eliciting tempting titilations.  Temperance!)

        Test 4 again suggests that no registers take longer to come up
than others.  In effect, it doesn't seem to take up any time to set up
a register.

        Test 5 verifies our intuition that reading $D601 doesn't alter
timings of subsequent operations.

        Test 6 is an interesting one.  A typical memory fetch takes a
fair amount of time!  Test 4 tells us that just setting up the register
doesn't take any time; it is the write to the registers, which then
causes VDC to go and fetch the data, that slows everything down.  This
will be discussed below.

        Test 7 shows that memory writes are somewhat time consuming.  Test 8
shows that the block-fill is a two order of magnitude improvement over
Test 7 though!  143 cycles versus 11000 -- wow!  Also of interest is that
there were some BIT-loop repetitions; I expected them to not be necessary.

        Andreas Boose provides a lot of insight into some of the issues
raised above:

>Stephen, 2MHz mode is not so easy as you might think. There are two facts
>you have to take in account:
```

>
>#1 In 2MHz mode a single raster line consists of 65 phi1 and 65 phi2
>cycles. But the 8502 doesn't get all cycles! During the last 5 phi1 cycles
>of any raster line the VIC-IIe must refresh the system's DRAM and so the
>8502 is switched to 1MHz on these cycles. Simply counting the cycles of
>the code and dividing them by 2 is not accurate. For a serious measurement
>you would have to synchronize the test loop to the VIC-IIe's refresh and
>to periodically count 120 cycles @2MHz and then 5 cycles @1MHz.
>
>#2 In 2MHz mode accessing VIC-IIe, SID, CIAs or $DE00-$DFFF forces the
>8502 back into 1MHz mode. But it is not simply done by counting these IO
>accesses as 2 2MHz mode cycles: It is different whether the IO access
>occurs during phi1 or phi2. If it happens on phi1 the VIC-IIe expands the
>cycle into phi2 and we get 2 2MHz cycles. If it hits on phi2 the VIC-IIe
>delays the IO request to phi1 and then the 1MHz mode access is performed.
>So this consumes 3 2MHz cycles instead the expected 2 cycles.
>
>For exact measurement 1MHz mode with switched off screen might be the
>best reference. But even with accurate 1MHz 8502 you will not get rid of
>odd values, the VDC has its own clocking frequency derived from a separate
>16MHz source which slides relatively to the 8502's system clock. Therefore
>there will be always some 1/2 or 1/4 cycle jitter on any $d6xx access.
>
>Also for exact measurement you have to align your routine to the VDC
>timing. In the moment you rely on the fact that the VDC can execute the
>same command in the same time regardless on what the VDC doing on the
>screen - but surely that matters. CBM engineers were 2MHz-DRAM-bandwidth-
>weenies and the VDC occupies lots of its RAM bandwidth for drawing the
>picture and refreshing the DRAM. So it must delay "foreign" RAM access of
>the 8502 to certain timing slots like the horizontal or vertical retrace
>or so.

        Clearly, my experiments have just scratched the surface of the VDC.
We can make a few conclusions however:

        - In many cases the usual BIT-loops are unnecessary. The things
which
          seem to make them necessary are the things which can tie VDC up,
          namely any operations which need to talk to his memory (did I
          mention that VDC is a 'he'?).

        - Some operations, such as a fill, are not only efficient but take
          a consistent amount of time to execute. Others, such as the
          memory fetch, do not.

The VDC strikes me as being 'the final frontier' of the C= world, in that
it is not terribly well understood and there is still much to be explored.
There are two things which I think would be very nice to have, concerning
VDC: first an understanding into the internal operation of the chip, and
second a list of timings for various operations involving VDC, with the
goal of really utilizing VDC for the graphics coprocessor that he can be.

```
Source + 2 binaries to follow.
--------------------------------------------------------------------------------

*
* vdctimes.s
*
* The purpose of this program is to gather some timing
* information on the VDC chip.  It simply uses CIA #1
* timer A to measure the time to perform a series of VDC
* operations.  A BASIC program will then statistically
* assemble the data.
*
* Stephen Judd
* 8/3/95

 ORG $1300

TIMALO EQU $DD04 ;Timer A, CIA #2
TIMAHI EQU $DD05
CIACRA EQU $DD0E ;Control register A

* Now some macros

SETREG MAC  ;Set a VDC register
 STX $D600
L1 BIT $D600
 BPL L1
 <<<

ELAPSE MAC  ;Calculate time elapsed
 SEC
 LDA #$FF
 SBC TIMALO
 STA ]1
 LDA #$FF
 SBC TIMAHI
 STA ]1+1
 <<<

 SEI
INITTIM LDA #$FF ;Stick 65535 into
 STA TIMALO ;timer latch
 STA TIMAHI
 LDA #%00011001 ;Start timer
 LDY #%00001000 ;Stop timer

CALIBRAT STA CIACRA ;Figure out overhead in
 NOP  ;starting/stopping timer
 NOP
 NOP
 NOP
```

```
 NOP  ;Elapse 10 cycles
 STY CIACRA
 LDA #$FF
 SEC
 SBC TIMALO
 STA TEST

* First test: No OP; just load a register

REGTEST
 LDX #18 ;Register 18
 LDA #%00011001
 STA CIACRA ;Start timer
 >>> SETREG ;A single reg
 STY CIACRA
 >>> ELAPSE,REG1
 LDA #%00011001
 STA CIACRA
 >>> SETREG ;Do it twice
 >>> SETREG
 STY CIACRA
 >>> ELAPSE,REG2
 LDX #36 ;Now we will do them all
 LDA #%00011001
 STA CIACRA
:LOOP >>> SETREG
 DEX
 BPL :LOOP ;Sub 5*37-1 cycles from total
 STY CIACRA
 >>> ELAPSE,ALLREG

READVDC ;Just a quick test to see if
 LDX #18 ;anything weird happens by a
 LDA #%00011001 ;read
 STA CIACRA
 >>> SETREG
 LDA $DC01
 >>> SETREG
 LDA $DC01
 STY CIACRA
 >>> ELAPSE,READ18 ;This should be REG2+8

WRITEVDC ;Various tests of writing to
 LDA #%00011001 ;the VDC
 STA CIACRA
 >>> SETREG ;Now read an actual memory
 STA $D601 ;location
 INX
 >>> SETREG ;Three setregs total
 STA $D601 ;Overhead: 6+6 cycles
 LDX #31
```

```
 >>> SETREG
 STY CIACRA
 >>> ELAPSE,READ31
 LDX #18 ;Now test memory fills
 LDA #00
 >>> SETREG
 STA $D601
 >>> SETREG
 STA $D601
 LDA #%00011001
 LDY #00
 LDX #31
 STA CIACRA
 LDA #$66 ;Screen code 102
:LOOP >>> SETREG
 STA $D601
 INY
 BNE :LOOP ;Overhead: 2+2+5*256-1
 ;(or 9*256 if sta is counted)
 LDY #%00001000
 STY CIACRA
 >>> ELAPSE,FILLSLOW

 LDX #18
 LDA #00
 >>> SETREG
 STA $D601
 INX
 >>> SETREG
 STA $D601
 LDA #%00011001
 LDX #31
 STA CIACRA ;Here we go...

 LDA #$66
 >>> SETREG
 STA $D601
 LDX #24
 LDA #%00000000
 >>> SETREG
 STA $D601
 LDX #30
 LDA #$FF ;Total of 256 writes
 >>> SETREG
 STA $D601 ;Off it goes!
 LDX #18
 >>> SETREG ;Hopefully this will wait until
 STY CIACRA ;the fill is done!
 LDA $D601 ;Just to make sure, check the
 STA MEMADDR ;last address written to
 INX
```

```
 >>> SETREG
 LDA $D601
 STA MEMADDR+1
 >>> ELAPSE,FILLFAST
 CLI
 RTS  ;Whew!

TEST DS 1 ;Calibration test
REG1 DS 2 ;Single register test
REG2 DS 2 ;Same register twice
ALLREG DS 2 ;All 37 registers
READ18 DS 2 ;Two reads of register 18
READ31 DS 2 ;Three reads minus 12 cyceles
FILLSLOW DS 2 ;256 writes
MEMADDR DS 2 ;lo/hi of memory fill (a check)
FILLFAST DS 2 ;One block write of 256 chars

--

begin 644 time1.0f.uu
M 1P7' $ F2 BFY-0051)14Y#12XN+B( )1P" $ZR,3(P.D.R, !6' , _B8Z
M1K(Q.I<@-3,R-C4LPB@U,S(V-2D@KR R,SDZCR!455).($]&1B!624, 91P$
M (\@1D%35#I&/3( H1P% (8@5#$H3BDL5#(H3BDL5#,H3BDL5#0H3BDL5#4H
M3BDL5#8H3BDL5#<H3BDL5#DH3BDL058H."D ]AP* %1%4U2RT2@B,31#.2(I
M.E(QLE1%4U2J,3I2,K)2,:HR.D%2LE(RJC(Z4C.R05*J,CI2-+)2,ZHR.D93
MLE(TJC(Z34&R1E.J,CI&1K)-0:HR #H=# !!5B@Q*1H:&AH:&AH:&AH:&AH:
M&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
M&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:&AH:
8&AH:&AH:&AH:&AH:&AH:&AH:&AH:

end

begin 666 vdctest1.0d.o
M !-XJ?^-!-V-!=VI&: (C0[=ZNKJZNJ,#MVI_SCM!-V-R12B$JD9C0[=C@#6
M+ #6$/N,#MTXJ?_M!-V-RA2I_^T%W8W+%*D9C0[=C@#6+ #6$/N. -8L -80
M^XP.W3BI_^T$W8W,%*G_[07=C<T4HB2I&8T.W8X UBP UA#[RA#UC [=.*G_
M[03=C<X4J?_M!=V-SQ2B$JD9C0[=C@#6+ #6$/NM =R. -8L -80^ZT!W(P.
MW3BI_^T$W8W0%*G_[07=C=$4J1F-#MV. -8L -80^XT!UNB. -8L -80^XT!
MUJ(?C@#6+ #6$/N,#MTXJ?_M!-V-TA2I_^T%W8W3%*(2J0". -8L -80^XT!
MUHX UBP UA#[C0'6J1F@ *(?C0[=J6:. -8L -80^XT!ULC0\J (C [=.*G_
M[03=C=04J?_M!=V-U12B$JD C@#6+ #6$/N- =;HC@#6+ #6$/N- =:I&:(?
MC0[=J6:. -8L -80^XT!UJ(8J0". -8L -80^XT!UJ(>J?^. -8L -80^XT!
MUJ(2C@#6+ #6$/N,#MVM =:-UA3HC@#6+ #6$/NM =:-UQ0XJ?_M!-V-V!2I
M_^T%W8W9%%A@                          :&AH:&AH:&AH:&AH:&AH:
1&AH:&AH:&AH:&AH:&AH:&AH

end


/S08::$dd00:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
```

Software analysis and reconstructive therapy,
a historical view on "cracking".

by Pontus Berg

Cracking was defined as the removal of copy-protection, and for this reason the
product had to feature protection in the first place! Most "originals" of today
do not feature any protection, but I guess what most people are after is just
introlinking and crunching (correct me if I'm wrong)! I'll brief you on the whole train of thought!

Cracking was much about being able to analyze someone elses code. This did not mean that you had to be able to understand exactly what each instruction did, but from the context you had to be able to tell the main purpose of a section of code. From this followed that you had to be able to know A LOT about coding. The better you knew how to code, the better was your chance of getting decent as a cracker. This was a very difficult thing, mind you. How could you detect a loader routine from the rest of the code if you didn't know 6502? The answer is that you simply couldn't! A lot of people had to learn on the fly. By digging into other people's code you got a feeling for 6502, and this experience built many a legend. If you ever saw a good cracker
in action they could scan the memory with the I* command of their monitor utilities and tell what parts of the memory did what and how. :)

This is all nice but what exactly went on in the mind of the cracker? Well, let's explore briefly the generalities on cracking tape games, as they are virtually non-existent in our Commodore world of 1996 and are therefore not a political hot-potato so to speak :).

For tapegames, the fact that the original is on tape is the hardest part. There are transfers that copies tapedata to disk for most common tapeloaders. Doing it by hand is not something you have to do every day, but it's something you must be able to do if you want to call yourself a cracker! Mind that it's not all that easy! If you take a look at the persons who call themselves crackers today, only a very limited few are able to do this!

Tapetransfer

SYS 63276

This loads the header into the tapebuffer at $033c. You get the name on

screen and can press space/CBM key to get back to the prompt. It won't start loading as it would have if you had written LOAD. You can now modify the tapebuffer. The five first bytes are 03 (Not relocatable file), Low/High startaddress, Low/High endaddress

f.ex. 03,9f,02,3c,03

This loads a file at $029f ending at $033c. If you modify this into

03,9f,12,3c,13

You'd get it on a place where it won't start in your face (i.e. $129F)! By the way, if the start says 03,01,08,XX,YY then the game loads to basic, and you can then just type LOAD and after loading it won't autostart (unless you pressed shift+runstop), but can be saved normally to disk (unless it exceeds the limits of basic by loading over $a000). Why $129F - Well as it's easier to disassemble the code if the lowbytes are the same as the original.

SYS 62828

Will load the file at the address you asked for with the pointers! (i.e. $129F)

From here you are on your own, as no two tape loaders are exactly the same! I cannot say anything general about tapeloaders! If you decidper the loader and find no startaddress in it, then it might be as it's overloaded during loading. Selfmodifying loaders are frequet to prevent you from tempering with them. I usually resource the loader and place it elsewhere in memory. I can then safely transfer the files!

The tapes don't usually feature any protections more that the fact that they are on tape. I seen a few games that set the timers before loading and checked them afterwards to ensure you hadn't interrupted the loading but This is rare! Last Ninja II was like this!

OK, after this you have the game on disk. A one-filer shall now work 100%, no matter if it's transfered from tape or if you got it directly on disk.

A onefiler

1. Depack it (if packed or crunched)
2. Remove crap that only take room and is of no use! (This is what I do GOOD that makes my cracks shorted than most of the others!)
3. Scan for trainers! (I use Action Replay first. If it finds them, I saved myself a lot of work!)
4. Install the trainers in a trainer menu & put it in memory of the game!
There's almost always room for that!
5. Pack the game
6. Add the intro

```
7.      Pack the game again (only needed if you gain something from this!)
8.      Crunch


Multifile games

Additional work compared to the above:

1.      Locate the loader and remove it
2.      Analyse the code for the old loader and make yours load the correct
level from the internal levelcounter of the game!
3.      Replace the loader with your own one (featuring a decruncher)
including
the information you got while analyzing the code (see 2)
4.      Crunch the levels


In detail

Ok you might then need some guidance in each of the steps! I guess
depacking is the most important part!

The depacker/decruncher starts somewhere (usually basic, at $0801), and you
can then pretty easily follow the code in it. The action replay has the
AWSOME feature "Set Freeze". You disassemble the code in freezemode (N.B.)
and then enter a SF the the opcodes.

F.ex.

4000 AD FF 4E  LDA $4EFF

You modify to:

4000 SF FF 4E  LDA $4EFF
     ^^
and press return!

You now get:

4000 20 D3 DF  JSR $DFD3

(Or something like this)

I saw QED/Triangle working, and he did a version of this using his Expert
Cartridge. He did pretty much the same but manually. He installed this
instead of the "SF"

CLC
BCC (To the CLC)

After a while, when the program was surely in the infinite loop, he slammed
```

```
RESTORE to enter his monitor. I have used this some times myself, when the
SF was not working for some reasons (mainly as $01 was set to something
that prevented SF from working).


Back to depacking

OK, back to the depacker! You enter this SF at the place where the
decruncher jumps to the next step in the process. It might be starting the
next depacker and it might be starting the game!

Remember that most decrunchers are different, so you cannot take for
granted that the skills from one apply on another one!


Some VERY GENERAL hints:

Most of the times, the startup looks something like this:

SEI/CLI/NOP     ;Either of these
LDA #$3X        ;Where X is almost always 4-7 (most often 7)
STA $01
JMP $YYYY       ;Where YYYY is the start address and the place to put your
SF
                ;Set on top of the $4C, i.e. the actual jump instruction.

Most decrunchers has this code in the area $0801-$0a00, but the AB cruncher
and CruelCrunch has it in the very end of the file (in the last two
blocks).

This is where I get to be so concrete that it would be best if I had an
example to show from!

This will have to be all for now.. :)

Pontus Berg


/S09::0100h:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::


                      - A Quick Overview of CP/M -

                            By Mike Gordillo


What is CP/M?
```

CP/M is an operating system available for microcomputers using the Zilog Z80 Microprocessor. As an operating system, CP/M is responsible for directing your computer's resources to fit your needs. It executes software (programs) in response to commands you enter through the keyboard (console), and once executed, your programs can use CP/M to perform many tasks. For example, CP/M software can manage databases, copy files, process spreadsheets, analyze your income tax, print your forms, or play your favorite video games.

CP/M 3.0+ on the Commodore 128 is simply a version of CP/M 3.0 that has been altered to better fit the particular needs of the Commodore 128 and its users. Aside from a few cosmetic differences and the ability to use the peripherals (drives, modems, etc) of the C-128 System, there is not much to distinguish general CP/M from the C-128 version. Your C-128 can execute virtually all the software widely available for CP/M machines. However, please note that CP/M software intended specifically for a certain CP/M machine may not necessarily run or run-as-intended on another CP/M machine.

How do I start CP/M?

Unlike the C-64 and C-128 native modes, CP/M must be started from the CP/M 3.0+ System Diskette. The standard start-up procedure requires you to have one drive set to device #8. All you do is insert the System Diskette into the drive and type BOOT from within the C-128's Basic 7.0 mode. CP/M should now "boot up" from the diskette. If this does not happen, clear memory by turning off the C-128 and the drive. Put the CP/M System Diskette in the drive and turn the C-128 and the drive back on. You should now see CP/M assume its normal diagnostic "boot up" procedure. Once that is complete, CP/M will announce itself proudly as "CP/M 3.0 on the Commodore 128" and provide you with its version date. The 28 May 87 CP/M 3.0+ version, for example, is the last and most popular official version for the C-128. Other "non-official" versions exist for the C-128, most notably the BIOS6-ZPM3-ZCCP enhancements -- but that's the subject of another article.

CP/M is a command-line interface!

CP/M is a command-line interface. This means that your instructions to CP/M must be entered as commands to the system from the keyboard (as opposed to mouse control in popular Windowing environments like GEOS). You enter commands via the A>, B>, C>, D>, E>, or M> prompts. These prompts tell you which disk drive (device) you are currently using as well as letting you know

that CP/M is ready to receive your orders as soon as possible. For example,
if I see the A> prompt, I am using device 8, but if I see the B> prompt, I am
using device 9, and so on. I can change between devices by simply entering the appropriate letter and a colon, as follows:

```
     A> B:                      (Changes from Drive A to Drive B)

     B> C:                      (Changes from Drive B to Drive C)

     C> A:                      (Changes from Drive C to Drive A)

     A>                         (Back to Drive A)
```

The E> and M> prompts are special cases. Drive E is not a real drive/device,
rather, it is a Virtual Drive used for copying purposes when only a single real drive is available. Drive M is not a physical drive at all. It is assigned to the Ram Expansion Unit as a RAM drive.

How is the CP/M Command Structure organized?

CP/M has six built-in commands: DIR, DIRSYS, ERASE, RENAME, TYPE, USER. These commands are part of your CCP.COM file (Command Console Processor) and they make it easy for us to perform basic level housekeeping. DIR allows us to call up a disk's directory of files. DIRSYS does the same thing except
it shows us only files tagged with a system-flag (eg. hidden files). ERASE and RENAME allow us to erase and rename files while TYPE is used to display text files on the screen. USER lets us switch between different user sections
on a diskette. A CP/M diskette can have 16 user sections (0 to 15) in which
files placed in one section are logically separate from files in different sections. User sections are CP/M's way of allowing for better organization of files through partitioning.

Any other "commands" you might run into are known as transient-utility commands. Transient-utility commands are simply executable programs that reside on a diskette instead of being built into CP/M itself. They are usually identifiable on the diskette by virtue of their .COM extensions. For example, if I call up a disk directory using the DIR command, as follows:

```
     A> DIR
     BASIC.COM     DISC=OV.ERY     ISAGREAT.MAG    SORT.COM    TERM.COM
```

This tells me that on Drive A (device 8), the files basic.COM, sort.COM, and term.COM are transient-utility command files. Notice that there is no

```
DIR.COM
available to call up a directory because the DIR command is a built-in
command.



Does my System Diskette have Transient-Utility (T-U) Commands?

    Yes, Commodore put several useful T-U commands into the CP/M 3.0+
System Diskette.   You should have the following in order to begin to
effectively use CP/M on the C-128, as follows:

     A> DIR
     C1571.COM    CONF.COM    FORMAT.COM    KEYFIG.COM    HELP.COM
PIP.COM

C1571.COM is a neat T-Utility that will double your 1571's SAVE speed
by disabling the automatic verification involving the saving of a file
to the diskette, as follows:

     A> C1571 [a               (Disables auto-verify on Drive A - Device 8)

CONF.COM is a GREAT T-Utility that permits us to change many system-default
parameters like baud rate, screen colors, printer assignments, etc.   When I
first started, I was very very frustrated at what appeared to be immutable
artifacts in the operation of the C-128's CP/M.   In other words, I could
not CONFigure the system as I wanted.    This utility changed all of that.
For example, I used to have a 4040 dual drive which I hooked up to the
system
intent on using it under CP/M.   I found to my dismay that I could only
access
one out of the two drives huddled in the 4040 bay.   CP/M had no way of
knowing
that the single device it saw (the 4040) had more than one drive!   Along
came
CONF.COM to the rescue!   I simply typed up CONF DRVB = 8-1 and voila, the
second drive in the 4040 became Drive B (which would normally be device 9).
CONF is also great at modifying CP/M itself when needed.   I recall one time
when I was using another T-Utility that had problems with the Ram Expansion
Unit under its normal device assignment.   In other words, the utility I was
using could not handle a drive M.   I used CONF's POKE ability to change
the drive M assignment of the R.E.U. into a drive B assignment, as follows:

     A> CONF POKE fbd3 = 96fb       (Reassigns the REU from drive M to B)

FORMAT.COM allows you to format your diskettes in a variety of CP/M formats,
all of which can be read by the C-128's CP/M.

KEYFIG.COM is a T-Utility which allows you to redefine ALL keys on the C-128
except the SHIFT and SHIFT-LOCK keys, the 40/80 DISPLAY key, the CONTROL
key,
and the COMMODORE key.   You can assign keys to have up to four different
```

values/functions to suit your needs.

HELP.COM is a good manual of CP/M commands and command syntax. It serves as
an index of commands and tells you how to properly use them.

PIP.COM is the standard file-copy T-Utility. I use it routinely to create backups of my files as well as creating new system diskettes in cooperation with the FORMAT.COM utility, as follows:

```
A> FORMAT             (Use the Format utility to format drive B)

A> PIP B:=A:CPM+.SYS  (Copies the CP/M system file from Drive A to B)

A> PIP B:=A:CCP.COM   (Copies the CCP.COM file from Drive A to B)
```

Is that all I can do with CP/M?

No, Commodore packaged CP/M for the C-128 with just the basic entry level material necessary to begin to use CP/M effectively. CP/M can do much more but like all things concerning computers, you need a good measure of software to make the best use of what you've got. There is nothing harder for the novice than to accumulate enough software to increase his and his system's proficiency. I recall many times thinking how useless CP/M was to me. After all, I could do so much more in C-64 and C-128 modes! However, once my software base had risen beyond what the System Diskette offered, my analysis of CP/M changed from useless to useful.

In other words, CP/M productivity is in software?

CP/M's strength lies in the large software base available for it. CP/M is the first operating system to showcase the functionality of microcomputer database and spreadsheet applications on a large scale. I regularly keep large databases and spreadsheets on the C-128's CP/M and then use other CP/M productivity programs to transfer them to and from other types of systems. Furthermore, some of the best software compression programs and computer languages are only available for the C-128 whilst in CP/M mode. With over ten thousand software titles to choose from, there is nothing you can't do!

Where can I get CP/M software?

You can get good CP/M public domain software from CP/M User's Groups throughout the United States and Canada. There is also substantial CP/M support in Japan and in Europe. Commercial software is readily available from appropriate vendors (usually listed in user group publications). The most substantial concentrated source of CP/M software is undeniably the Internet. The Internet is a collection of various computer networks linked together throughout the world. There are thousands of megabytes of CP/M

programs (from 1976-1993) available there. A few popular 'CP/M sites' are as follows;

```
            oak.oakland.edu     /pub/cpm
            ccnga.uwaterloo.ca  /pub/cbm/os/cpm
```

And pray tell, how do I transfer CP/M software into the C-128?

Well, there are a number of ways. The most efficient way is to copy files from one CP/M disk onto another. The C-128's CP/M can read the CP/M diskettes from Epson, IBM, Kaypro, and Osborne CP/M machines. You can also use terminal programs if your CP/M's version-date is later than 4 DEC 85 (ie., earlier versions lacked access to the modem port) in order to transfer material via the phone lines. The most inconvenient way, however, is to use transfer programs (from either CP/M or other modes) to read material
onto your CP/M diskettes.

Come on, I'm looking for something simple. Is there a CP/M CD-ROM?

Sure is! A few people out there decided to put several megabytes of CP/M software on CD-ROM (!) and just a few months ago, someone wrote a CD-ROM reader for C128 users with a CMD Hard Drive (SCSI port)! The "Walnut Creek" CD-ROM and "CD-ROM Commander" are two very exciting products for CP/M 128 users, as these items allow access to hundreds of megabytes of CP/M software. (Please consult the internet newsgroup 'comp.sys.cbm' for more details).

Final thoughts...

CP/M complements the C-64 and C-128 modes very well. I feel that in the productivity area, the C-128 would be very deficient otherwise. I have yet to see productivity packages in the native modes that are as efficient as
the ones I use under CP/M. The structure of the CP/M operating system is also more amenable to file transfer and file management. My Ram Expansion Unit has never been as easy to work with in the C-64 and C-128 native modes as it has been under CP/M. CP/M may seen complicated at first but it is actually quite easy to understand. Once you've mastered it, you may never go back.

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

```
/S10::0100h:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::


      The BIOS-R62a/ZPM3/ZCCP Commodore 128 CP/M 3.0+ Upgrade Package
                      and a bunch load of utilities!

                              by Mike Gordillo


                                   Preamble

    This package was an in-house project for mostly myself and a few of
my friends.  I released this last year to the general internet community,
and as far as I know, all software therein is public domain.

What to do:

FTP to ccnga.uwaterloo.ca
Directory is /pub/cbm/os/cpm

Download (remember to set binary flags) the following:

        zpm-user0.zip
        zpm-user1.zip
        upgrade-user0.zip

Unzip them on your unix/vms/dos box and put them on a C128 CP/M
1571 or 1581 boot disk (eg., BBR 4.1) in the following manner:

All files from zpm-user0.zip go to USER 0 on your CP/M boot disk.
All files from zpm-user1.zip go to USER 1 on your CP/M boot disk.

IF you have any of these: Swiftlink cartridge, an ASCII printer,
absolutely NO ram expansion whatsoever, a 1581 drive, you will probably
need to take a look at upgrade-user0.zip.  It contains replacement files
that satisfy the hardware I just listed.  IF so, you MUST use the
replacement files to insure proper operation.  All replacement files MUST
go to USER 0 on your CP/M boot disk.  Please READ the read.lis file in the
upgrade-user0.zip archive for more details.

Note: If you unzip these archives on a VMS system, chances are that
CPM+.SYS will be renamed to CPM.SYS.  You MUST make sure that you rename
it back to CPM+.SYS on your CP/M boot disk, otherwise CP/M will not boot.

For more information on BIOSR6 and ZPM3/ZCCP enhancements please consult
Randy Winchester's CP/M article in the online magazine, C= Hacking issue #5.
```

Abstract

This package is provided as a courtesy to C128 users. Meaningful CP/M system-generation utilities are not provided with the C128 CP/M System diskette. Most C128 users would therefore have difficulty in assembling a package like this. Also, the lack of easily available sources for CP/M software brings its own share of hardships. The utilities included are meant to provide new tools for the C128 user and maximize the features of BIOS-R62a, ZPM3, and ZCCP.

Filelist

zpm-user0.zip:

Length	Date	Time	Name ("^" ==> case conversion)
512	01-03-95	11:41	autotog.com
4480	01-03-95	11:41	b5-driv3.com
512	01-03-95	11:41	bye.com
128	01-03-95	11:41	c128-xgr.z3t
1024	01-03-95	11:41	c1571.com
3200	01-03-95	11:41	ccp.com
24576	01-03-95	11:41	cpm+.sys
1152	01-03-95	11:41	echo.com
3840	01-03-95	11:41	format.com
16384	01-03-95	11:41	format22.com
640	01-03-95	11:41	format81.com
3328	01-03-95	11:41	if.com
3456	01-03-95	11:41	loadseg.com
5376	01-03-95	11:41	mkdir32.com
256	01-03-95	11:41	names.ndr
17536	01-03-95	11:41	qterm.com
17408	01-03-95	11:41	rdcbm.com
8192	01-03-95	11:41	salias.com
3456	01-03-95	11:41	setpth10.com
1024	01-03-95	11:41	startzpm.com
8192	01-03-95	11:41	superzap.com
24320	01-03-95	11:41	trans128.com
7424	01-03-95	11:41	v.com
15488	01-03-95	11:41	vde.com
2816	01-03-95	11:41	verror.com
15744	01-03-95	11:41	vlu.com
2048	01-03-95	11:41	zdt12.cfg
7936	01-03-95	11:41	zdt12.com
15232	01-03-95	11:41	zfiler.com
1536	01-03-95	11:41	zinstal.zpm
217216			30

```
zpm-user1.zip:

 Length    Date    Time    Name ("^" ==> case
 ------    ----    ----    ----   conversion)
    128  01-03-95  14:47   clrhist.com
   3328  01-03-95  14:47   conf.com
   5996  01-03-95  14:47   conf.hlp
   3072  01-03-95  14:49   copy.com
   3200  01-03-95  14:49   date.com
   2816  01-03-95  14:50   del.com
   3712  01-03-95  14:51   diff.com
   3712  01-03-95  14:51   dir.com
   1280  01-03-95  14:52   dirnames.com
   2944  01-03-95  14:52   diskinfo.com
   1664  01-03-95  14:53   image.com
   6912  01-03-95  14:53   lt.com
  12928  01-03-95  14:55   pmext.com
   3712  01-03-95  14:58   ren.com
   1792  01-03-95  14:58   rsxdir.com
   4736  01-03-95  14:59   unarc.com
  12288  01-03-95  15:00   unarj.com
   1408  01-03-95  15:02   undel.com
   3456  01-03-95  15:03   unzip.com
 ------                    -------
  79084                    19


upgrade-user0.zip :

 Length    Date    Time    Name ("^" ==> case
 ------    ----    ----    ----   conversion)
  24576  01-03-95  15:41   cpm+.sys
   3840  01-03-95  15:40   f1581.com
    256  01-03-95  15:41   names.ndr
  17536  01-03-95  15:46   qtermsl.com
    732  01-03-95  14:08   read.lis
   1024  01-03-95  15:41   startzpm.com
 ------                    -------
  47964                    6



                              Condensed History

BIOS-R62a- Default System Baud Rate set at 134.  Warning: Term programs will
           modify this.  Higher Baud = Faster Keyboard Scanning = Slow CP/M
           Re-implemented support for PETSCII printers (code from BIOS R4)
           LST Settings : PRT1=Dev #4, PRT2=Dev #5, Secondary Address = 7
           CONF utility's PRT assignment options will not work because of
           changes made back in BIOS R4.   ASCII printers are available with
           the ASC-PRT implementation of BIOS-R62a (included..see BIOS R5).
```

```
           -CPM+.SYS for ASCII printers is included in upgrade-user0.zip


BIOS-R62 - Default System Baud Rate set at 75 (not enough Keyboard
Scanning).
           Added support : C=1581 Official Format! (F1581.COM will allow you
           to make/create 1581 boot disks... included in upgrade-user0.zip).
           MAXI 71 and GP 1581 Format supported (see BIOS R5).
           ASCII printers still default.
           -Randy Winchester
BIOS R5  - Added support for new hardware: Quick Brown Box (E:), Drive D:
           Added new definitions to the disk-parameter-table.  Maxi 1571,
           GP 1581 formats.  Use format22 & format81 with these MFM
           types.  ASCII printers now default.  PETSCII tables not
supported.
           LST Settings : PRT-D4=Dev #4, PRT-D5=Dev #5, Secondary Address =
5
           -Randy Winchester

BIOS R4  - Removed the 40 column routines, the virtual drive, and Drive D:.
           Removed Printer Buffer (Lord knows why!?)
           Added a screen dump feature.
           Fixed several BIOS errors as well.
           -James Waltrip IV

ZPM3 BDOS- (see below)
           -Simeon Cran

ZCCP CCP - (see below)
           -Simeon Cran


                                  **DISCLAIMER**
**
**You are free to distribute this package with the following conditions;
**
**   A) This package cannot be sold.   A copying/handling
**      fee of no more than $5 1993 US dollars is allowed.
**
**   B) This package shall remain whole.   No item may be
**      distributed apart from the rest of the package.
**      There is a degree of hidden cross-dependency between
**      some items.   Split them apart and you may get
**      unpredictable results!
**
**This package is NOT under any warranty or guarantee of ANY kind!


                                  Description
```

```
   BIOS Upgrade - The C128 28 May 87 CP/M 3.0+ Version BIOS was reworked to
                  remove useless code (40col screen and Virtual Drive...not
                  really useless to some of us?  Argghhhh! :) and to correct
                  a few CP/M 3.0 BIOS errors while adding a screen dump
feature
                  (ALT key is used as a toggle).   End Result = Faster, more
                  "peppy" CP/M 3.0+ operation.

   BDOS Upgrade - This is the ZPM'ing of CPM!  Think of this as a way-
overdue
                  correction for an anachronism.   Much of the original BDOS
                  is written in slower Intel 8080 code.   The ZPM3 BDOS
upgrade
                  rewrites things in faster, richer Zilog Z80 code while
                  adding some goodies (eg., command history buffer, enhanced
                  command line editing, automatic command prompting) and
                  correcting some CP/M 3.0+ BDOS errors.

   ZCCP Upgrade - The last nail in the coffin.   The original CCP.COM is
                  replaced by a more flexible beast.   Neat things are now
                  at your beck and call.   ZCCP features :

                      ZCPR 3.3 Compatibility (see below)
                        -Does not support FCP but supports flow control
                           internally with an IF.COM utility present.
                        -RCP is not implemented (That's what REUs are for :)
                        -Cannot load ZCPR 3.4 "type 4" programs
                        -Cannot re-execute loaded programs sans re-loading
                      Z3T Termcap (ZCPR 3.3 graphics support)
                      Named User Groups/Directories
                      Command Search Path
                      System Environment Block
                      Flow control processing for batch files
                      Extended Command Processor for batch files
                      Multiple commands on one line
                      Superior error handling
                      Up to 4 Shell stack levels may be defined
                      Direct loading of .RSX files without GENCOM (LOADSEG)
Put these all together and you have the ultimate CP/M system for your C128.



                                Compatibility

BIOS R62a - 99.00% Compatible with stock CP/M 3.0+ C128 system.
                 -A problem concerning printing is listed in the
                  Condensed History section.
            (Note: Programs that call the 40col screen will see a NULL40
            label -i.e., They will run but they won't be able to do anything
            in 40cols)

ZPM3 BDOS - Fully 100% compatible with stock CP/M 3.0+ BDOS segments.
```

```
            (Note: Some rare programs *demand* the CP/M 2.2 BDOS..yuck!)

ZCCP CCP  - ZCPR 3.3 compatibility as seen in the Description section.
            Environment info is larger than before.   Slightly more TPA is
            used or some additional high memory is being toyed with.  For
            example, I shortened TRANS128's buffer and that did the trick.
            (Note: Some of Steve Goldsmith's C128 specific programs will
             will crash with the ZCCP.RSX in operation).

                                Additional Notes

    -Look at the ALIAS (included) batch file (type SALIAS STARTZPM on the
     command-line).   Notice how it optimizes the system for REU use.
     You may change this as long as you keep the following in mind :

       ZCCP *requires* : LOADSEG commands for NAMES.NDR and *.Z3T Termcaps.
                         At least *ONE* SETPTH drive search/path entry.

       Quick Summary of STARTZPM batch file (included):
           1) Loads up Directory names.
           2) Loads up Directory paths.  ($$$$ = Current Drive/User DIR)
           3) Executes a few .COM files.
           4) Copies all Command Utilities to M1: (COMMAND Directory).

    -Utilities copied over to the REU (as seen in STARTZPM) are general
     purpose utilities meant to replace the built-in commands of the
     standard CCP.COM.  With the excellent path setups in ZCCP, utilities
     in the speedy REU become transparent.

     (Note: If you have NO ram expansion (eg., a drive M:) you NEED to
     take a look at upgrade-user0.zip, as mentioned earlier).

    -Multiple commands on the command line must be separated by semi-colons.
     (Note: Semi-Colons are used in CP/M 3.0+ to append file passwords. Use
the
     SET.COM utility (SET [DEFAULT=PASSWORD]) to set a password which can be
     used without your intervention on every file you access.  In any case,
     you can assign passwords to user groups (under ZCCP) with the mkdir32
     utility, which is simpler than dealing with CP/M 3.0+ SET.COM password
     assignment schemes.)

    -The following keys have been already configured to work best with ZCCP:
     They are user-definable with KEYFIG or LOAD/SAVEKEY utils (not
included).

    -CRSR-UP/DOWN - CRSR-LEFT/RIGHT  KEYS
     UP = CTRL-E   DOWN = CTRL-X   LEFT = CTRL-S   RIGHT = CTRL-D
     CTRL-E and CTRL-W = Forward and Backtrack through Command-History
Buffer.
     CTRL-X = Delete everything to the left of the cursor.
```

```
    -ARROW KEYS (at the top of the keyboard)
     UP = CTRL-E   DOWN = CTRL-X   LEFT = CTRL-A   RIGHT = CTRL-F
     CTRL-A and CTRL-F = Autotab left and right.

    -CLR/HOME = CTRL-H (BackSpace)
     INST/DEL = CP/M RUBOUT KEY

    -ZCCP does not support printable control characters (eg., ESC, CTRL-Z)
     on the command line.  In order to change screen display characteristics
     use the CONF.COM utility (included) instead of, for example, using
     ^[^[^[<screen/char color code>.   Also, although CTRL-Z will not clear
     the screen anymore, you can use the built-in CLS command instead.
     (Note: You can still use printing codes in programs.)

    -Consult the C128 system manual for the full list of ADM-3A to C-128 key
     assignments and sequences.

--
For general comments on this article, feel free to contact Mike Gordillo
at s0621126@dominic.barry.edu


::::::::::::d:i:s:C=:o:v:e:r:y:::::::::::::::::i:s:s:u:e::1:::::::::::::::::
::
\H01:::::::::::::::::::::::::H A R D W A R
E::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
```

BEYOND STEREO

The POWER-SID Model 2400 : It May Not be THX, but it's
Taking the Commodore One Dimension Closer

A preview by Shaun Halstead

A few months back, as I was working on some music using Robert Stoelle's Stereo Sid Editor, I got to thinking about how many voices I would need for this and that, and I realized that I was going to need more than twice as many as I had available. I called up Nate Dannenberg (a.k.a. _Tron_) to see if he knew of any way to get more voices; he said that the only feasible way was to add more chips, of course. So, being the annoying pest I am, I got to thinking of ways to do just that, and the concept of surround sid board was born. Carrying eight, count 'em, eight, sid chips, at three voices each, combined to give four main channels (front and rear, with left and right on each), working in tandem with the computers stock sid chip, which acts as a center channel, the Power-Sid is carrying the Commodore into a new

```
dimension
of sound.

    Well, to make a long story short, between the two of us, we developed
the basic components, layout, and operation, including addressing options.
Currently, the board is designed to mount vertically, similiar to the
Super-CPU developed by CMD.  This was done mainly to conserve space behind
computer, but has another advantage: by mounting the card vertically, a
pass-through port is easily installed (it was a given from the start that
a pass-through be available).  Until recently, we were having considerable
difficulty in solving the pass-through mounting problem (i.e. where to put
it), until Nate found a way around it, based on the Super-CPU.  The actual
construction will not be easy, by any means, but it should be quite strong,
durable, and nice in overall appearance.

    The model described here (model 2400) will measure roughly 4-by-4
inches, contain eight sid chips, two stereo jacks, addressing and support
circuitry.  The support circuitry includes an enable/disable switch,
activity
LED, an addressing switch, selectable between $DE00 and $DF00, and a pair
of LEDs indicating the current address, a two/four-channel selection switch
and dual-color LED to indicate the current mode.  The only difference
between
the model 2400, described here, and the model 1200 is that the 1200 carries
only 4 chips, and offers fewer voices per channel, but is none the less,
fully
featured.  Currently, software is being developed by Nate and I, and we are
working on a port to support his up and coming digital output card for the
expansion port.  Look for an extensive follow up in this journal when the
board enters the prototyping stage.

--
The author, Shaun Halstead, can be reached at tesla@onyx.southwind.net,
and Nate Dannenberg can be reached at tron@wichita.fn.net, or catch them
on IRC channel #C-64, as _Tesla_ and _Tron_, respectively).


\H02::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::


                    *  The 8 bit Modplay 128 Board  *
                                  by
                          -- Nate Dannenberg --


Have you ever listened to Digital sound on your Commodore and wondered
where all the noise is coming from?
```

Ever tried messing with Pulse Width Modulation, only to realize that it just doesn't have the clarity and impact you need for a particular project?

Want to make your Commodore sound better than it ever has using 8 bit sound?

Well, now's your chance...

Included here is the procedure for building a two channel 8 bit DAC circuit to plug into a Commodore 64 or 128 (works in both modes), which outputs in stereo two-channel sound. Actually, the 8 bit DAC chip we will be installing does four tracks, which are mixed using four 1K resistors into two channel stereo sound with a total resolution of 9 bits per channel, or 8 bits per track.

This circuit is based around the Maxim Electronics MAX505xCNG chip, where x refers to a number of chip-packaging options provided by Maxim. The recomended chips are the MAX505 A- or B- CNG (24 DIP Narrow Plastic package).

This chip contains four 8 bit DAC circuits, each with 5 volt rail-to-rail swing, internally buffered and coupled with precision unity-gain followers that slew at 1V/uS (If you can understand all that jazz, you're a better man than I). Each DAC accepts it's own reference voltage input (not to exceed the power supply voltage, which is 5 volts in this project), and each generates it's own buffered output.

Originally, this circuit was designed to work on a Commodore 128 with the upcoming MODplay 128 software, also one of my projects. Using this software, and outputting to this circuit, I get about 8.2 Khz out of it in four track, two channel, 8 bit-per-track (9 bit total) stereo.

Since this circuit is relatively simple to access, it can be used in many other applications, from audio output from a simple RAW or WAVE player type of program, to more complex applications requiring a controllable analog signal. With the right timing, you could probably use it to control the sync signals to an RGBI monitor!

Anyways, let's get on with it.

First we need to get the legal stuff out of the way...

Disclaimer: I take no responsibility for any loss or damages occuring from the use of this device, or from the instructions and proceedures described in this text. Any damage resulting from the use of this text is purely the responsibility of the builder/user.

[Ed. Note : The editor nor disC=overy magazine is not reponsible for any loss
 or damages occuring from the use of this device, or from the instructions and
 procedures described in this text.]

```
Here is the parts list:

1) Maxim Electronics MAX505ACNG (or -BCNG) DAC chip.
1) 12/24 .156" Female edge connector.
1) 1/8" stereo (three-conductor) miniplug.
4) 1K, 1/8 or 1/4 Watt resistors of either 5% or 10% tolerance.
*  a bit of thin wire (preferrably wire-wrapping-wire)
*  a bit of heavier wire (about 20 guage)
*  solder (silver-based is preferred, but lead based works too)
*  A low-wattage soldering iron (mine is only 20 watts)

Optionally, you may build this onto a printed circuit board, which will
require a piece of double-sided copper clad board about 1" by 2.5", a
resist-ink pen or dry transfers, and some Ferric Chloride (PCB etchant).

[Ed. Note : Maxim Electronics : 1-800-998-8800 ; Radio Shack has the jack
 and they do carry a .156" connector but with 22/44 leads.  I had to cut
 it to 12/24 in order to match the userport.  Also, it must be noted that
 soldering to a socket is less of a risk than soldering to the chip itself.]

I built the prototype by mounting the chip directly to two pins of the
user port connector.  Since the design is so simple, I will describe the
procedure for building the circuit using the prototype method, without
using a printed circuit board.

Here we go:

1) Turn the edge connector to point the solderable pins towards you.  Cut
   off pin 1, pins 3-7, and pins 10-12.  These are on the TOP row, with
   pin 1 being on the left end when the connector is held as above.

     left   1   2   3   4   5   6   7   8   9  10  11  12   right

2) Cut off pin A and B.  This is on the bottom row, second from the left.
   Remember that as with most Edge connectors, there are no pins G or I.
   Instead the pins are lettered like this:

     left   A   B   C   D   E   F   H   J   K   L   M   N   right

3) Align pins 12 and 13 of the Max chip, with pins H and F (respectively)
   on the User Port plug.  Make sure the chip and plug are right side up.
   now bend the chip's pins around and under the plug's pins, and solder.

4) Using a series of short wires, connect the following using ordinary
   point-to-point soldering.

      Chip   Plug
       16      C
       15      D
       14      E
       13      F  (already soldered)
```

```
       12      H  (already soldered)
       11      J
       10      K
        9      L

5) Using more point to point soldering, connect the following:

      Chip   Plug
       22     two
       19      M
       18    nine
       17    eight
        8      N

6) Turn the device upside down and make the following connections on the
   back side of the chip.  Be sure not to confuse your pin-layout!
   Again, plain old point-to-point soldering is the key.

   Pin 20 to pin 21, Pin 21 to Pin 22, Pin 22 to Pin 4, Pin 4 to Pin 5.
   Pin 8 to Pin 7, Pin 7 to Pin 6, and Pin 6 to pin 3.

7) Turn the circuit back right-side-up again, and connect one end of each of
   the four 1K resistors, to the Max chip, pins 1, 2, 23, and 24.

8) Now Take the resistors connected to pins 2 and 23, and connect the free
   ends together.  Connect this point to the Miniplug TIP tab.

9) Take the resistors connected to pins 1 and 24, and connect the free
   ends together.  Connect this point to the Miniplug SLEEVE tab.

10) Connect the Max chip's Pin 6 to the Miniplug GROUND tab.

11) Take a piece of the heavier wire, fashion a simple pull-handle by
    running a wide loop of it between the holes at each end of the plug.
    For strength I recommend soldering the ends together.

[Ed. Note : Please observe : Pins 1 and 24 are used as RIGHT output and pins
 2 and 23 are used as LEFT output.  On a 1/8" stereo miniplug jack from
Radio
 Shack (#274-249a), you will see three prongs.  The prong closest to the
 opening of the jack is the GROUND, and for reference's sake we will
consider
 that this prong is at the base of the jack.  The other two prongs are on
 the back of the jack and are aligned one 'above' the other.  The one that
 is on the base is the SLEEVE and the one above it is the TIP.]

Your 4 channel DAC circuit is now ready to use.  To test the circuit, you
can run this short BASIC program on a Commodore 64, or a Commodore 128 in
64 or 128 mode.  In 128 80-column mode, you may want to type 'fast' and
hit return before running, to make the test higher pitched and a bit more
accurate.
```

```
10 u=56577: poke u+1,63: poke u+2,100: input "channel (1-4)";c$: c = val(c$)
20 if c<1 or c>4 then 10
30 if c=1 then ch=248
40 if c=2 then ch=252
50 if c=3 then ch=240
60 if c=4 then ch=244
70 poke u-1, ch
80 for x=0 to 255: pokeu,x: next
90 get a$: if a$="" then80
100 goto 10
```

What you should hear coming out from the desired channel is a sawtooth wave, simialar to the sawtooth save the SID chip produces, but of course a tad * distorted * since this is in BASIC. Press any key to stop the test.

In machine language, these instructions should produce a similar test.. POKE 250,speed: POKE 251, chan_index: SYS 3072.. speed will control the pitch of the sound. Chan_index is 248 for channel 1, 252 for channel 2, 240 for channel 3, and 244 for channel 4. Press the spacebar to stop the test.

```
        Hexadecimal     Decimal equivalent

START   * = $0C00;      3072
        SEI
        LDA #$3F;       63
        STA $DD02;      56578
        LDA #$FF;       255
        STA $DD03;      56579
        LDA $00FB;      251
        STA $DD00;      56576
        LDY #$00;       0
LOOP    STY $DD01;      56577
        LDX $00FA;      250
DELAY   DEX
        BNE DELAY
        INY
        LDA $DC01;      56321
        CMP #$EF;       239
        BNE LOOP
        CLI
        RTS
```

Accessing the board via your own software is very simple..

You need to use Machine code if you are planning to access more than one channel of the board. If you plan to use a single channel, just POKE 56576,248 and run your player routine. This will select channel 1 (left side) to send your output through.

In machine code, you use a single LDA:STA pair to select the channel, and just stuff your data into $DD01, something like the above.

The hardware will take care of the output, and will see to it that the byte you send out is properly clocked and that it goes to the correct channel.

The User port contains a total of 10 easily programmable data lines (the others require more complicated methods to use). These lines are broken into two sets. The first set, the data bus as I like to call it, consists of the 8 lines than make up the User Port's PB0-7 area. These bits are all present at location $DD01. The data direction regiter for this location is at $DD03. A 1 bit there means an input, while a 0 bit signifies an output.

The other set consits of two bits that sit at location $DD00, these bits are PA2 and PA3, also known as RS232 data out, and Serial ATN in, respectively. These two bits serve as a mini-address bus, and are used to select the correct channel to comminicate with.

Since the hardware inverts the output of PA3, it was necessary to re-invert this bit in software, which is why the channels are being selected with such odd numbers.

Also, since location $DD00 controls the 16K memory bank seen by the VIC-II chip, you must keep the VIC pointing to bank 0 (by setting bits 6 and 7) if you want to keep the VIC 40 column display in proper order. This explains why all of these numbes are 240 (#$C0) or higher.

Most DAC chips require some sort of clock signal to pass data into the chip. The MAX505 is no exception. In order to lessen the time needed by the computer, we wil ise the User Port's PC2 line, a low active clock line that fires every time a byte is sent or received via the PB0-7 lines.

This line was intended to drive the /FLAG input of another C64, or other hardware device, however, it also serves as a useful clock signal for our chip.

The MAX505 requires you to select the channel first, write the data byte to it's data bus, and then pull the /WR line low for a brief moment. Thus we write our code like this, to do these actions in proper order:

```
LDA #$Cn            ; n is 8, C, 0, or 4, for Channels 1, 2, 3, and 4
                    ; respectively.  Channels 1 and 4 are wired for left
                    ; output, while 2 and 3 are wired for right.
STA $DD00           ; This actually sets the channel number.
LDA your_data_here  ; Use whatever method you need to get the next byte of
                    ; Sample data into the accumulator.
STA $DD01           ; Send the byte to the User port, automatically
                    ; firing PC2.
```

That's pretty much it!

The MAX505 chip is designed to handle in excess of 100,000 samples per second, per channel. In fact, according to the specs, it can go as high as 1 million samples per second, per channel.

Below you will find a pinout diagram for the MAX505 chip, to aid in performing the steps given above for assembling this circuit. This is a duplicate of the diagram found on page 1 of the spec booklet that comes with the MAX505 family.

```
             TOP VIEW

            ___  ___
           Ü.  À-'   Ü
  VOut B [ 1Ü         Ü24] VOut C
  VOut A [ 2Ü         Ü23] VOut D
     VSS [ 3Ü         Ü22] VDD
  VRef B [ 4Ü  MAXIM  Ü21] VRef C
  VRef A [ 5Ü MAX 505 Ü20] VRef D
    AGND [ 6Ü         Ü19] A0
    DGND [ 7Ü         Ü18] A1
   /LDAC [ 8Ü         Ü17] /WR
(MSB) D7 [ 9Ü         Ü16] D0 (LSB)
      D6 [10Ü         Ü15] D1
      D5 [11Ü         Ü14] D2
      D4 [12Ü         Ü13] D3
           Ü_________Ü

           DIP/SO/SSOP
```

Have fun with this circuit! I built mine in about 30 minutes, and it has become my default playback device for my Modplayer. I will also write in full support for this device in Sound Studio 4.0 (commercial), as well as any other digital sound programs I happen to write later.

The MAX505 chip will also be used in another board I am designing, called the E-8 (Enhanced-8) board. This board will use two MAX505's to provide an effective output resolution of 16 bits per channel. The board will also feature a pair of National Semiconductor ADC0820BCN chips, for sampling in 8 bit two channel stereo (Unless I find a better chip to use). No release date or pricing has been set for this board as of yet.

--
For more information on this or general commentary, you may reach Nate at the following addresses:

Dannenberg Concepts, Ltd.
"Bringing your Commodore 128 one step closer!"

Email: tron@wichita.fn.net
Phone: (316) 777-0248

Snail: Dannenberg Concepts, Ltd.
         c/o Mr. Nate Dannenberg
         306 S. 1st Ave
         Mulvane, KS   67110
FidoNet: 1:291/25
         Groups: CBM, CBM-128, and CBM-GEOS
Usenet: comp.sys.cbm, alt.binaries.sounds.mods

\H03:::
::
::
::

Upgrading your C128's VDC memory to 64K

by Mike Gordillo

This will be a quick "how-to" on upgrading your VDC memory from 16K to 64K. Those of you who have C128-DCR's should already have this upgrade from the factory, but those who own original flat C128's or original C128D's may wish to undertake the following VDC ram expansion. First off, the last time I checked, you needed two 64x4 DRAM's and the prices for such ram chips were as follows:

   LA Trade - 1-800-433-3726
   64x4 DRAM - 80 nanoseconds = $3.75

   Nevada Computer - 1-800-982-2925
   64x4 DRAM - 100 nanoseconds = $2.45

   COMP-USA !!! 1-800-COMP-USA
   64x4 DRAM - 100 nanoseconds = $1.50 ! ! ! !

The instructions for installing them into your machine quite simple. Just open up your C-128, remove the shielding, keyboard connector, etc. You should
now see a metallic box around the center of the C-128's motherboard. Remove the lid on that box with the keyboard facing you, revealing the following:

>>>>>>> Back of C-128 <<<<<<<

```
  VDC   -  6   -     --**--             /   VIC Chip
  Chip  -  3   -                        /   Circuitry
        -      -     ------     ------  /   to the right
        -  V   -     - 16 -     - 16 -  /
        -  D   -     -  x -     -  x -  /
        -  C   -     -  4 -     -  4 -  /
        -      -     -    -     -    -  /   <---These two 16x4 DRAM's
        -      -     -DRAM-     -DRAM-  /       are to be replaced with
        --------     --**--     --**--  /       64x4 DRAM's.

                          ** = designates the directional notch on the chip

>>>>>>> Front of C-128 <<<<<<<
```

In your machine you will see two 16 by 4 DRAM chips (16 kbytes total). Your task is to replace them with 64 by 4 DRAM chips (64 kbytes total) that you can
get cheaply at CompUsa for example :)...no minimum order.. :)). The hard part
is now at hand. The 16 by 4 chips are not on stacks! This means you have to
unsolder them. If you have never played with a soldering iron before this is
going to be a BIG pain in the rear and you may damage your C-128 in the process. (See, isn't it neat to find out for FREE instead o paying Slick Wile
up front?)

If you have some experience with a soldering iron, remove the C-128's mother board from the case, flip it over, and find the pins for the 16 by 4 DRAMs on the back. Unsolder them and remove the chips. Solder the 64 by 4 DRAM chips in the same exact spots as the 16 by 4 chips, remembering to keep the notch on the 64 by 4 chip facing the notch on the original 16 by 4 chip.

Solder in stacks/sockets first, and then plug in the 64 by 4 chips, because it is safer to solder in the sockets AND if a memory chip goes bad, you won't
have to unsolder again (just pop out the ram chip from its socket.)

18-Pin Stacks/Sockets are 50 cents each at Radio Shack.

DISCLAIMER :

You undertake this procedure at your own risk. If you destroy your machine, its your cross to bear. I did this to a C-128 last Sunday. It took 2 hours with the simple soldering tools I have. It may take you less time or more time
depending on your equipment and experience. DRAMs are a pain to remove in

any
case. I sped up removal by just cutting the legs on the original 16 by 4 chips
with a manicurist's scissors.

Definitions :

16 by 4 (16 x 4) = 8 Kbytes of memory
64 by 4 (64 x 4) = 32 Kbytes of memory
DRAM = Dynamic Random Access Memory
8563 VDC = The C-128's 80 column video chip.
LC = Logic chip next to VDC, more specifically, its a 74LS244.

Specific labelling on the original 16 by 4 DRAM:

You should either see TMS4416 on them or MB81416. They are usually 120 nanosecond chips, so you must buy 64 by 4 chips that are at least 120 nano-second variety. I chose 100 nanoseconds because it was the least expensive 64 by 4 DRAM speed that I could find.

--
For general comments on this article, feel free to contact Mike Gordillo at s0621126@dominic.barry.edu

\H04::
::
::
::

>> The Metal Shop ---
+
--- with SMS Mike Eglestone <<
of Diamondback BBS
(305)258-5039

Senior Master Sergeant Mike Eglestone has been a devout C= Hardware guru for over ten years and is the Sysop of one of the most active BBS's on the Commnet Commodore BBS network. Mr. Eglestone has written for illustrious magazines in the Commodore 8 bit community such as dieHard magazine and the very distinguished Commodore World. We are pleased to have him entertain our questions pertaining to hardware repair. As always, neither the editors,
the staff of disC=overy, nor Mr. Eglestone are responsible for the use or misuse of any information presented in this article.
--

```
>> Dear SMS Mike,
>>
>>    My CMD Hard Drive refuses to undergo its power-up sequence at
irregular
>> intervals and is belching out a whole series of errors and bad blocks.
>> Unfortunately, my drive is long past its warranty period and I have no
>> idea what is wrong with it.  What do you think?
>>
>> MG

MG,

 First of all, I would give CMD a call about this even if your drive is out
of warranty. They know their drives better than anyone, but with that said,
I have run into a total of 5 (was 4) Hard Drives produced by CMD that had
cold solder joints on the main circuit board. They were discovered at the
connection point for the Power Supply, and were all on the 12VDC connectors
for the drive motor.

 There is a 4 pin DIN plug attached to the main circuit board which is the
plug for the power supply to the drive. The connection on the bottom of the
board (remove it and turn it over) has a tendency to be bad or loose from
poor solder joints (cold joints).

 This cold or loose joint will cause strange problems with the drive motor,
and you will notice SCSI errors, or a sudden accumulation of Bad Blocks. It
can even result in the complete failure of the drive to start up, or it
might
just shut down (stop turning) without warning.

 The solution is very simple. Remove the drive and circuit board. Turn the
board over, and re-solder the joints. Takes about 15 minutes from start to
finish...

 The way the main circuit board is attached to the case is one of the
problems. The rear support point is in front of the DIN plug but close to an
inch away from it. Each time you insert the plug into the DIN connector, it
puts pressure on those connection points and BENDS the board slightly. After
a
while, the connections loosen up at the solder joints. Cold Joints or joints
that were just slightly tacked, will crack or just seperate from the foil.

 I have now fixed 5 drives with this same problem, and they work perfectly
again... CMD is aware of the situation, but they think it's an isolated
problem with one batch of boards. I don't think so. I think it's going to
happen to all of them sooner or later. That rear support and the location of
the DIN seem to be a failure point.
--


>> Dear SMS Mike,
```

>>
>>   I am about to throw my C128's keyboard out my balcony. For the past
>> year, a key has been failing to work properly about once a month. I now
>> have to put up with at least 12-15 keys that either refuse to work at all
>> or require some deep-felt pounding in order to generate a character.
>>
>> What can I do about this aside from buying another keyboard??
>>
>> SK

SK,

 I have cleaned and restored hundreds of keyboards over the last 10 years. I have piles and piles of boards and board parts (64 and 128) sitting around my computer/repair room. Not to mention around 10 extra mother boards for both
types of computer.

 After playing around with the carbon contacts on both the circuit board itself, and the carbon impregnated (rubber contacts) on the end of the key plungers, I have come to the conclusion that the only cleaner required is de-natured alcohol. It works on every part of the board, and works perfectly.

 When I do a cleaning job, I remove (strip) every key down to its component parts. The keys themselves go into a bleach solution. The rubber pads and plungers go into a pan of denatured alcohol. The key springs (if not rusted) are lightly sprayed with WD-40 and rinsed with alcolol later. If they are rusted, they go into a solution of phosphoric and dichromate acid. Brand name
is OSPHO (any True Value Hardware Store has it).

 The circuit board itself is washed with de-natured alcohol and lightly dried
with a blower.

 Seldom do the carbon spots on the circuit board go bad. If they do, you can use a product which contains silver loaded epoxy, and mix that with carbon powder; Graphite. There is also a spray carbon compound that's available, but
I don't remember the name at the moment. It's used by the aircraft industry.

 Once, I couldn't find any silver loaded epoxy. I just used a hobby type (two
part) epoxy and mixed in huge quanitities of graphite. It worked perfectly. Dab on a drop, let it setup, lightly sand the surface to break the glaze.. (wham), a new conductive contacting surface.

 It was trial and error for a long time, but I now have it down to a science.
I normally get four years out of an overhauled keyboard; using original

parts
and proper cleaning solutions.

 Oh sure, I have written off a few keyboards as not worth fixing, but that isn't the norm. When one comes in that has been abused to the point that I don't even want to attempt a repair, it becomes parts for other boards.

 One thing always needs to be done with keyboards (prior to re-assembly).

  Plug in the circuit board, turn on the computer, then touch each keypad to a contact (one by one) and make sure it's working. This is done by hand.

 Blank un-assembled (open) circuit board, with keypads held in the fingers. It takes about 5 minutes, and it's well worth the effort involved.

 NO electrical voltages are present, that will harm you. There are only a couple of volts of DC present, and that is the output of the CIA (Complex Interface Adaptor) which is the Keyboard control chip.

 You can't run a conditivity test on the rubber keypads with a tester. It's got to have a slight DC voltage present to make it work. A standard VOM will not do the job. The computer itself makes a dandy test bed.
--

>> Dear SMS Mike,
>>
>>   Every now and then I'll be sitting peacefully in front of my 128 when
>> I start to hear crackling noises out of the monitor's speaker and I'll
>> notice that the 40 column screen starts to go wacky on me at the same
>> time. Sometimes I'll see color changes and sometimes I'll see little
>> random characters appear on screen. The crackling noises appear to
always
>> be a consistent low pop-pop-click and occur in both 64 and 128 modes.
>> A few rare times, 128 mode will crash and hang when these events occur,
>> although 64 mode will be a-ok (except for the crackling and popping).
>> I know my monitors and cable are working 100% so I am at a loss to
explain
>> this phenomena.
>>
>> RR
--

RR,

 There are three possibilities, and one of them is going to sound a bit odd.

 The 128 uses a two stage video output system. The VIC chip and a seperate Vidio Controller. Both are located inside the metal Box (with lid) in the back
left hand side of the circuit board. There is a screw hole dead center of the

metal box.. The VIC chip would be my first choice. It's on the right hand side
of the box. IC-U21 is the number on the board. The Vidio Controller is mostly
for RGB displays and 80 col memory enhancement, but it does work WITH the VIC
chip in some area of Composite color IC-U22 (Left hand side of the box).

Here is the "odd" choice, but one that I have found in quite a few 128 video
and "Crackling Sound" problems.. -> The ON/OFF switch for the 128 itself.

As you are aware, the 64 and 128 use a dual voltage power supply system. The
computer has the second stage on the Circuit board. That On/Off switch can become dirty inside, or limited in movement enough so that it doesn't make contact perfectly in the ON position. Sometimes it's just a matter of flipping the switch on HARD to make good contact, and sometimes it's necessary to actually remove the switch and clean the contacts internally.

On 6 different 128's, where I have come across this problem, I have had to remove the circuit board and File out the switch Hole "larger" at the top because the computer CASE itself was interfearing with the movement of the switch. A "not quite closed" contact in that switch will cause the 9VAC section to ARC slightly. This will drop colors and cause the sound you were talking about. It will affect your internal clocking and cause both video and
audio problems.

It's always best to check the IC's by substitution. Take a known good IC and
replace the one in use. If that doesn't solve the problem then do the same thing with the other chip. Sometimes, just pulling the chip out and putting it
back in will fix contact problems. Humidity will cause minor corrosion on the
legs of an IC over a period of time. Re-seating the IC will normally fix this
sort of thing. A quick shot of contact cleaner in the IC socket never hurts.

Those are your three choices guy.

Well, there is ONE more, but it's a remote possibility. The DIN socket on the
composite video OUT side. You can give that socket a shot of contact cleaner and see if the problem clears up. It will sometimes build up a slight film and
cause similar problems to the one you are talking about.

Last thing. If you remove the RF shield (the metal cover over the circuit board) LEAVE it OFF. The computer will run cooler and you really don't need

```
that hunk of metal. I normally throw the darn things away. Although it is
used
as an RF shield and a heat sink, it causes more problems than it solves.

 SMS MIKE
--
Mr. Mike Eglestone may be reached for questions or comments at Diamondback
BBS
(305)258-5039 or through the editors of disC=overy.


\END:::::::d:i:s:C=:o:v:e:r:y:::::::::::::::::i:s:s:u:e::1:::::::::::::::::
::
::::::::::::::::::::::::::::::May  17,
1996::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::
```