Issue #21
Feb 5, 2002

Special Focus on Minigames

.....

"Do not meddle in the affairs of wizards, for they are subtle and quick to anger." -- Ancient Elvish saying

.....

BSOUT

(Why yes, I _am_ re-reading Lord Of The Rings, how did you know? Sure, the movie is pretty good, not the same as the book, but probably the best you could do, although why does every Hollywood evil creature essentially come from Night of the Living Dead? But hey, the elves actually looked like elves. Black Hawk Down is really good too, and... oh, sorry!)

Hoo-ah! Welcome to another issue of C=Hacking!

There's lots of nifty stuff to get to so this will be brief. First, the "Hacking Exchange" is now up at the Official Unofficial C=Hacking Homepage:

http://www.ffd2.com/fridge/chacking/

It's just a simple message board, with the idea that C=Hacking types can make comments, ask questions, and otherwise talk to one another. Check it out!

Second, if you have any projects you're working on... please contact me, and consider writing them up for C=Hacking!

And finally, something I think you'll enjoy:

http://groups.google.com/groups?hl=en&th=87c2a2ced7e32ce1&rnum=42

From: duck@clumsy.pembroke.edu (duck@clumsy.pembroke.edu)
Subject: C= Hacker's Net-Mag
Newsgroups: comp.sys.cbm
Date: 1992-02-08 14:52:20 PST

Due to the recent influx of "Tech-Know" (People who actually understand / hack te C=64 into doing stuff that was previously unknown) I am going to tentativly start a C= Hackers Net-Mag (Hacking in the 90's since of the word, not the 80's).
So - please send me your netmail address if you're interested in receiving it.
Or if you are interested in contatcting me concerning an article etc that you'd like to see distributed please also contact me.

For the first issue I'm tentatively planning on:
- Line Drawing on 8563 VDC (640 x 200 hi-res graphics)
- Beginning ML column
- Raster Article
. . .

- Craig Taylor
duck@pembvax1.pembroke.edu

P.S. - Not sure when the first is gonna go out but hopefully soon.

From: duck@clumsy.pembroke.edu (duck@clumsy.pembroke.edu)
Subject: Issue 1 - C= Hacking Available
Newsgroups: comp.sys.cbm
Date: 1992-02-27 17:22:44 PST

Issue 1 of C= Hackers is now available via NETMAIL and is a compilation of several articles on the tehnical side of the Commodore 64 and 128. For those of you who missed the first posting, please reply via email and ask to be put on the list. The first issue had programming in ml, documented and undocumented 6502 opcodes, and a line-drawing package in machine language for the C=128 hi-res screen.
Issue 2 will be coming out in a month or so. Many thanks for the bandwith.

- Craig Taylor
duck@pembvax1.pembroke.edu

....
....
..
.

C=H 20

:::::::::::::::::::::::::::::::: Contents ::::::::::::::::::::::::::::::::

BSOUT
- o Voluminous ruminations from your unfettered editor.

Jiffies
- o News, rumours, and stuff.

Side Hacking

- o "Pulse Width Modulation, continued" by various.
  Tying up some loose ends from last issue's digi article.

- o "Introducing Full-Screen IFLI mode with a SuperCPU", by Todd Elliot <eyethian@msn.com> (Hey, what's with all this MSN crapola? :) Using a SuperCPU, it is possible to use the first three columns of an (I)FLI picture, and Todd shows how.

Main Articles

- o "VIC-20 Kernel ROM Disassembly Project, part IV", by Richard Cini <rcini@msn.com>

  And now it's time to start on that most frightening of creations: the tape drive code!

- o "The Art of the Minigame" -- an article in six parts:

  Introduction, by the editor
  Part 1: Codebreaker, by David Holz <whiteflame52@yahoo.com>
  Part 2: TinyPlay, by S. Judd <sjudd@ffd2.com>
  Part 3: MagerTris, by Per Olofsson <magervalp@cling.gu.se>
  Part 4: Compressing Tiny Programs, by S. Judd <sjudd@ffd2.com>
  Part 5: TinyRinth, by Mark Seelye <mseelye@yahoo.com>
  Part 6: Tetrattack!, by Stephen Judd <sjudd@ffd2.com>

.... Credits ....

Editor, The Big Kahuna, The Car'a'carn..... Stephen L. Judd
C=Hacking logo by..... Mark Lawrence

Special thanks to the cbm-hackers for many otherwise unacknowledged contributions.

For information on the mailing list, ftp and web sites, send some email to chacking-info@jbrain.com.

.... Jiffies ....

$01 Jochen Adler has made a program that reads the second side of a 1541 disk in a 1571 - without turning the disk over. It reads the blocks from end to beginning. Because of the mechanical bump however, it can only read tracks 5 to 35. If anybody wants this program please e-mail Jochen (NLQ@gmx.de)

$02 Soci/Singular has been working on a commented C128 ROM listing. Check out this great effort at http://singularcrew.hu/c128rom/

$03 64net/2 has been updated:

For those that do not know: 64net/2 is yet another PC-to-C64/128 User<->LPT parallel cable software. It supports d64/d71/d81/t64/lnx/dhd disk images, raw files and own Internet partition. It is possible to enter disk images like any other directory. Client programs are provided for C64 and C128. The next goal will be to patch ROM instead of loading client.

A small BASIC example program is included that is able to send e-mails through 64net/2 host or any other machine if its IP is known. How many e-mail agents are there for C=? How many of them are written in plain BASIC 2.0? :)

http://sourceforge.net/projects/c64net/ contains the latest version, and

http://venus.wmid.amu.edu.pl/~ytm/64net2win.tar.gz contains a Windows binary.

$04 CC65 is up to version 2.7.0, with many new improvements. Check it out at http://www.cc65.org/

$05 Moreover, Ullrich has set up his C64 as a web server at

    http://c64.cc65.org/

The web server runs on a stock 64, using a Swiftlink for communications, and uses the uIP TCP/IP stack written (with cc65) by Adam Dunkels

    http://dunkels.com/adam/uip

Pretty cool, eh?

$06 A new IDE interface has become available:

Elysium is proud to announce a new software and hardware solution for your Commodore mass data storage needs.

The CIA-IDE is yet another approach to connect an IDE hard drive to C64/128. It differs from previous similar projects in these areas:

- it is the simpliest one (only two chips required (or just one in case of C128))
- it is free (documentation and software),
- the software is available in source code form under GNU GPL,
- there _exists_ a ready to use GEOS 2.0 driver.

Documentation, source codes, and binaries are at:

    ftp://ftp.elysium.pl/groups/Elysium/Projects/ciaide/

$07 Marko Makela has developed a tape drive emulator with an RS-232 port, allowing transfers between any computer with a tape port and any computer with an RS-232 interface (e.g. a PC or Swiftlink). The hardware and software is at

    http://www.funet.fi/pub/cbm/crossplatform/transfer/C2N232/

$08 GoDot -- the C64 image processing program -- is now public domain. Arndt will still be working on it, but it's now available at

    http://members.aol.com/howtogodot/godnews.htm

$09 The C64 is now listed in the Guiness Book of World Records; a scan of the page (from Robert Bernardo, posted by Frank Michlick) is at

        www.cupid.de/upload/famous.jpg

(love that line about "16K sound").

Also, I highly recommend taking Robert's advice and checking out the infinitely cool "Logo-Matic" on the main www.cupid.de site!

$0A JOS just keeps cruising along, with lots of changes. Among the biggest changes, of course, is the announcement that JOS will be merged with Clips, with the new system to be called "Wings" (with some of the letters capitalized and some not; I never remember that stuff). For the latest JOS news, check out

http://www.jolz64.cjb.net/

$0B Frank Kontros has made a commented disassembly of the C64 ROM, with the BASIC ROM on the way:

http://c64.uz.ua/sources/C64_Kernal_Disassembly.zip

$0C And finally, Aleksi Eeben, author of two minigames, has now written a VIC-20 game:

http://www.student.oulu.fi/~aeeben/download/dragonwing.zip

more screenshots:

http://www.student.oulu.fi/~aeeben/screen1.png - screen4.png

Aleksi's homepage is at http://www.cncd.fi/aeeben

Neat!

..... Side Hacking .....

Pulse Width Modulation, continued
---------- from various

The digi article in issue #20 of C=Hacking left a few loose ends, and generated some followups.

First, Otto Jarvinen (sounddemon) emailed to say that the SID detection routine occasionally reported incorrect results for him, and suggested that a workaround was to do the detect several times. YMMV!

Second, a day or two after issue #20 was released, Levente discovered a brilliant way to play 6-bit PWM digis on a stock machine:

--
I couldn't resist, and tried something out (see attachment). It works!!! :-)

In fact, when I wrote the last letter I didn't know that I found something useable, just had some ideas - I felt that I'm at the right place. When I read C=H 20 this morning and read your comment about the Test bit (from the PRG), I knew that it must work. All I had to do is then to put this idea into code.

The whole idea is about starting the pulse by software, and then having the SID turn it back to 0 after a time.

Is it possible? ...The keys are the Test bit (the SID wave counter can be reseted anytime), the pulse width register, the wave counter and the SIDs way of generating pulse wave. (Ie. the pulse wave is high, as long as the wave counter is less than the value in the pulse width register).

Check this algorithm:

- Init: volume at max, voice 1 sustain level max, start attack. Freq is selected well (=$4000), so the wave counter is incremented by 4 every processor clock cycles.

Loop:
- load next sample value, and put it to the pulse width low register ($d402; ensure that $d403 is 0).
- Set test bit, and clear test bit (counter reset).
- Increase sample pointer, some delay, then loop. The delay must be 64 clock cycles + the time while the Test bit is kept set (4 cycles if using STA $d404 : STX $d404 immediately with pre-loaded values).

What will happen? The 8-bit sample value is put directly to the pulse width register (MSBs of the pulse width register are cleared!...). The wave counter is started (release test bit), and it increases 4 by every CPU cycles (= counts 256 in 64 cycles). After some time, the counter will reach the value in the pulse width register. This happens in exactly after (8-bit sample value / 4) cycles, because of the above. In this cycle (or the next?...) the SID turns its pulse output to 0. Voilá!

One must just make sure that the loop length in cycles matches the above conditions, and then it runs like hell... Since it does exactly the same on the SID as the other (bit-banging) way, it just does it with some hardware help, there's also no problem with the 4khz maximum barrier (since the

oscillator is reset every loop).

With little enhancement, it's possible to write an about 7.5 bits player for a stock C64 by this method. This is what you find in the attachment... The idea is using all the 3 channels simultaneously. A slightly increased sample value is written to the three pulse width registers, so the oscillators will finish the duty cycle one processor cycle later, when there's a carry between bits(0,1) to the MSBs.

The replay freq is the CPU clk / 68 (~15khz). 64 cycles (variable duty cycle) + 4 cycles (constant duty cycle because of the reset time - no problems with that, it doesn't change (just gives a small constant DC...)).

By similar methods, it should be possible to write a sample player with higher PWM freq (with less resolution of course, but eliminating this still audible whistling).

(I tried using the filter to reduce it, but it sounded so bad that I left it out. It clicked like hell. The FETs got saturated.)

[Richard Atkinson suggested turning down the sustain volumes to avoid this]

See the attachment, and the binary. I think the sample sounds pretty good :-). (The cut is from 'Greece 2000' by Three drives on a vinyl).

(Another idea that popped up in my mind: since the TED sound generator can also be reset, I could probably translate this idea to the Plus/4 :-O).

Best regards,

Levente

--

The binary is available at http://www.ffd2.com/fridge/chacking/ towards the bottom of the page.

Third, I received a very interesting email from an Apple-II guy, which I'd like to pass on:

--

Hi!

I found your page as I was searching for something else 6502-related, and was very interested. Although I have always been aware of the C64, I have never really been a user--I have used Apple II's since 1980.

I was particularly interested in the article on playing "digis" on the C64. I became interested in playing digitized sounds on the Apple II in 1993, after hearing a 3-bit, 11.025 KHz PWM player. At 3 bits, you can imagine how noisy speech samples were, but the overall effect for a 1 MHz machine with a 1-bit speaker "toggle" was amazing. It made me wonder how far this PWM technique could be pushed on a stock, 1 MHz Apple II (not the somewhat faster, 65816-based IIgs).

The short answer is, much farther than I expected! Robin and Stephen accurately describe the theoretical PWM limit as 6 bit samples at about 16 KHz for a stock 1 MHz machine, but, as they point out, that is not practically realizable for a number of reasons, unless the play loop is completely unrolled!

Furthermore, in the Apple II world, sampled sounds have acquired a few standardized sampling rates--mostly as a result of Mac influence, which was in turn influenced by CD's. The most common rate in the Apple II world is 11.025 KHz, or one-fourth of the audio CD sampling rate. This is commonly considered to be "AM radio quality", with a Nyquist bandwidth of about 5.5 KHz and a practical bandwidth of 4+ KHz, given practical anti-aliasing filters (at the sampling end, not the playback end).

A frequency of 11.025 KHz is, though high, still painfully audible to people whose ears are not zonked--a piercing "squeal" running through every sound. So even though it is possible to write a practical 6-bit 11.025 KHz PWM player (usually called a SoftDAC in the Apple II world), the resulting listening experience is disappointing.

So I went to work on a way to do 2x oversampling, and built a 5-bit 22.050 KHz PWM player. It was sad to lose a bit, but the absence of any audible "carrier" more than compensated for it!

If you have access to an 8-bit Apple II (preferably with lower case, like a //e), and also preferably with a way of attaching an external speaker or headphones in place of the miserable 2.75" internal speaker, then you can easily give it a try and judge for yourself.

I'm pretty proud of the novel design of the code, which I would characterize as "vectored" unrolled loops, one for every two pulse duty cycles, which I wrote a BASIC program to write for me--much less painful for counting cycles!

The package is available on the web at:

http://members.aol.com/MJMahon/index.html

and is called <A HREF="http://members.aol.com/MJMahon/sound22.shk">Sound Editor v2.2</A>, since I had to "dress up" the player into something fun to play with. ;-) An earlier version of Sound Editor was published on SoftDisk in 1994, IIRC, but this one is a little more evolved. It also introduced 2:1 ADPCM compression of 8-bit sampled sounds, to save disk space. It is a lossy compression, but not very noticeably. The editor package also includes those routines, in 6502 assembly code.

All of this should be trivially adaptable to the stock, 1 MHz C64, with very good results. By using the filters, you could probably filter out the 11.025 KHz carrier and return to 6-bit accuracy!

I should note that in the Apple world, sampled sounds are usually represented as "excess-128" codes, which means that the sign bit is inverted. This actually simplifies things, since the sample value is within a few shifts of being the pulse width in cycles.

Let me know what you think!

-michael

--

(Always great to hear from Atari and Apple][folks!)

And finally, I have a little mathematical analysis of PWM and how it compares to a "straight" digi. Basically, I found some of the PWM explanations a little unconvincing in issue #20 (even though I wrote them!). For example, the idea of "average voltage" seems a little funny, since every two samples has an "average voltage", as does every four, etc. but that set of average voltages would give a different sounding signal than the original (or more dramatically, there is an average voltage over a full second of digi playback, but that's not what you hear!). So I wanted to know how a PWM signal _really_ compares to a straight digi playback.

Another issue is changing the amplitude of a PWM digi, i.e. using two pulse waveforms, with one 1/16 the value of the other, to get higher resolution. If you recall the discussion of digis, the resolution of a PWM digi depends on the number of pulse widths available, not the amplitude. Adding two PWM waveforms together does not change the number of pulse widths available, so I wanted to figure out what changing the amplitude _really_ does to a PWM digi, and if it can really be exploited.

And finally, I wanted to know about the carrier wave (that is so piercing at lower playback frequencies) -- and once again, how it compares with a standard digi (which, after all, is stair-stepping the voltages at the playback rate).

Since the rest of this article is some Fourier analysis that 99% of people will have zero interest in, I'll put the conclusions here. The first is: PWM digis and standard digis are essentially identical except at higher frequencies (except for a phase shift, which doesn't make any difference to your ear). The second is: changing the amplitude of a PWM changes the resolution. More specifically, the amplitude of the pulse multiplies the digi sample value. If two pulses can be synced close enough, it should indeed be possible to use two pulses to get a higher resolution. Moreover, by modulating the amplitude of a single PWM digi, using the $d418 volume register -- that is, using PWM _and_ $d418 -- it should be possible to get a higher dynamic range, something that should be a little more achievable using SID (but maybe not that useful, so I didn't try it out). And finally, a standard digi has zero amplitude at the carrier frequency.

In other words, after a lot of effort I was able to demonstrate what everyone already knows.

The analysis doesn't change anything from the previous articles (except

possibly the idea for changing the PWM amplitude to get more dynamic range).

And now, some Fourier analysis. A standard digi just sets the voltage to the sample value s_j, for a length of time dt (dt = 1/sample rate). The Fourier transform of a single sample s_j (occuring at time t_j) is

$$s_j [e^{(-iw dt)} - 1] * [e^{(-iw t_j)} / -iw]$$

where w = angular frequency. Since the above is a little hard to read, I'll say it in words. The first term is the sample value s_j, which scales amplitudes at all frequencies. The second term is due to the finite length of the pulse (evaluating the Fourier integral at the boundaries), and basically changes the phase of the transform. The third term is like sin(w)/w -- a sinusoid with decreasing amplitude as frequency increases. So: the transform goes like sin(w)/w times the sample value, with some phase effects thrown in (we'll get back to these in a moment).

A PWM digi sets the duty cycle of a pulse to the sample value s_j, giving a Fourier transform of

$$[e^{(-iw s_j dt)} - 1] * [e^{(-iw t_j)} / -iw]$$

Compare this with the earlier expression, and you'll see that the sample value s_j has moved up in to the exponent of the "phase term" but that they're otherwise the same.

The first thing to do is to show that both expressions, PWM and standard, reduce to the same thing -- that is, that a PWM and a standard digi sound the same! The expressions both decrease as 1/frequency, due to the sin(w)/w term. This means that at large frequencies the values become negligible. (How large? For example, if the sample frequency is just 1KHz, then sin(w)/w is .001 times smaller near w=1KHz (i.e. the sample frequency, which is twice the Nyquist limit) than it is near w=0).

So now consider the phase terms for small w. The Taylor expansion for e^x is

$$1 + x + x^2/2 + ...$$

We can therefore expand the "phase terms" as

$$\text{regular: } e^{(-iw dt)} - 1 = (1 - iw*dt + w^2 dt^2/2 + ...) - 1 = -iw*dt + O(w^2 dt^2)$$

$$\text{pwm: } e^{(-iw s_j dt)} - 1 = -iw*s_j*dt + O(w^2 dt^2)$$

where O(w^2 dt^2) is considered very small since w and dt are both small. Substituting the above into the original expressions gives

$$s_j*iw*dt [e^{(-iw t_j)} / iw]$$

in both cases. That is, we have shown that for "small" frequencies -- more specifically, for frequencies where (w^2*dt^2) is much smaller than (w*dt), which is where w*dt<1, which is frequencies less than the sample frequency, which is all frequencies of interest! -- PWM and standard digis are the same.

The explanation lies in the phase terms. Those "phase terms"

$$[e^{(iw dt)} - 1] \quad \text{(regular)}$$

and

$$[e^{(iw s_j dt)} - 1] \quad \text{(PWM)}$$

do more than just change the phase. When they multiply the sin(w)/w signal, they take the sin(w)/w signal, change the phase, and then subtract the sin(w)/w signal again. It's this difference of signals that makes things work out at the frequencies we care about. PWM and standard digis are _not_ the same, but the main differences are at higher frequencies, where the amplitudes are in general much smaller.

But... but... what about the PWM carrier frequency? If we take a constant digi, say with sample values = 1/2, the standard digi gives a constant voltage, whereas a PWM digi gives a square wave at the sample frequency. The answer comes from the "phase terms" above. The sample frequency is

$$w = 2*pi/dt.$$

Substituting this into the phase terms gives

[e^(i*2*pi) - 1] (regular)

and

[e^(i s_j 2*pi) - 1] (PWM)

The regular expression is exactly zero -- there is _nothing_ at the sample frequency of a regular digi. But that's not the case for the PWM term, because of the s_j up in the exponent. PWM digis have a _finite_ amplitude at the carrier frequency. Note that because of the sin(w)/w term it gets smaller as the sample frequency increases -- but it isn't zero.

Finally, the phase term expansions give some insight into what happens when both the pulse width _and_ height are varied. If the pulse width is s_j, and the height is set to h_j, then the Fourier transform becomes

h_j*s_j *iw*dt [e^(-iw t_j) / iw]

That is, the amplitude multiples the width. For the case of adding two PWM waves together, then, the amplitude really does effectively scale the sample value, and it should be possible to add one PWM value at 1/16 the amplitude of another to get an effective 8-bit value.

What about _varying_ the amplitude of a single PWM sequence? For a 6-bit PWM digi, say, the sample values s_j can go from 0 to 63. If this is then multiplied by h_j=2 say, then the values become 0 2 4 ... 126 -- a 7-bit number where the lowest bit is always 0. What use is that? Well, we still have the h_j=1 values of 0..63, which do include the lowest bit. So we can effectively change the dynamic range from 0..63 to 0..126 using just two amplitude values.

As a practical matter, then, it might be possible to use all 15 $d018 values available to get a big dynamic range, and hence a better sounding digi, using fewer CPU cycles. Well, ok, we're only _sort of_ changing the dynamic range, so I pretty much doubt the usefulness of it. But maybe someone out there would like to give it a shot.

All right, let's hope this closes the book on pulse width modulation for digi playback!
....
....
..
.

C=H 20

.....

Introducing Full-Screen FLI mode for the SuperCPU

The 'FLI Bug', where the first three columns of a FLI screen are essentially unusable, can be squashed with the help of a SuperCPU. I won't go into great detail on IFLI, as it has been well-documented elsewhere, but I'll begin with a short summary to get us all up to speed. I refer you to Albert 'Pasi' Ojala's excellent coverage of the FLI mode in C=Hacking #4. Pasi also proofread this article.

A Three-Minute Summary of the FLI mode

The VIC-II chip asserts a badline when it needs to access the databus and fetch character data or videomatrix data. It was discovered that the VIC-II chip can be manipulated by its vertical scroll register at $d011 (SCROLY) to induce a badline at any given rasterline. By having a badline at every visible rasterline, the program can manipulate $d018 (VMCSB) to point at the right videomatrix to achieve the maximum flexibility of colors given to a multi-color screen.

Unfortunately, when a program forces a badline via SCROLY, the BA (Bus Available) line in the computer goes high, and for three cycles the 6510/8510 processor has to finish its write operations or halt its read operations before the BA line is released to the VIC-II chip. The maximum number of successive write operations is three, hence the 3-cycle delay. It is in those three cycles that the VIC-II does not fetch video matrix data to fill in the first three columns and causes the 'FLI Bug'.

I wish to stress that in those first three cycles, when the BA line is high, the 6510/8510 processor is still active and can complete write operations. It isn't fully shut down. After the badline retrigger at STA SCROLY, the code following it is fetched on the databus and is ready to be executed by the 6510/8510 processor. When BA is high, the VIC-II will reference the value on the databus as videomatrix data and display it in the first three columns of the screen. The actual instructions that follow the STA SCROLY in the FLI

loop constitutes the video matrix data for the first three columns of the screen.

Enter the SuperCPU!

Normally, a VIC-II chip access is only possible every 4 cycles. The SuperCPU can access the VIC-II chip in 1 cycle (1MHz) intervals, making cycle to cycle changes possible within the VIC-II chip. More importantly, the SuperCPU tristates the 6510/8510 processor inside the host Commodore computer (which is a fancy way of saying that you can disconnect the processor from the system without physically removing it).

When a forced badline retrigger occurs with a STA SCROLY in a FLI loop under the SuperCPU, the BA signal inside the host Commodore computer goes high. But, the SuperCPU runs asynchronously and really doesn't have to pay attention to the host Commodore as it runs code after the STA SCROLY. In fact, the SuperCPU will execute code even if the VIC-II badline is in full swing inside the host Commodore computer.

I knew that the instruction opcodes left on the databus after the STA SCROLY made up the video matrix data for the VIC-II chip for those first three columns of the screen. But I wondered how this was possible in a SuperCPU configuration because there would be no instruction opcodes left hanging on the databus inside the host Commodore computer. After some discussions with Per Olofsson ("MagerValp"), he suggested that writes/reads to the i/o area will force a value to be put on the databus.

This is where the magic begins, when the FLI loop forces the SuperCPU to write to the i/o area of the host Commodore after the forced badline retrigger at STA SCROLY. The SuperCPU will note that the BA signal is still high, so it can still access the databus and stash values there via DMA. This BA high signal will last for 3 cycles, enough for the SuperCPU to stash three values onto the databus.

The 6510/8510 is still tristated by the SuperCPU, and there's nothing on the databus after the forced badline retrigger at STA SCROLY. Normally, the 6510/8510 CPU shares the databus with the VIC-II for each machine cycle. With the 6510/8510 CPU out of the equation, the SuperCPU can stash a value onto this shared bus on the CPU half of this machine cycle and the VIC-II chip will see it in its other half of the machine cycle.

However, the databus is only eight bits wide. The VIC-II chip fetches video matrix data and color ram data 12 bits at a time. The SuperCPU can force values onto the databus during the first three cycles after the forced badline retrigger, but on each cycle the last four bits belonging to Color RAM would not be fed to the VIC-II chip. Only pixel values of %10 and %01 can be individually selected in multicolor FLI mode, while %11 pixel values cannot be individually set for those first three columns of the screen. The high resolution FLI mode does not suffer from this problem because it does not use color RAM for color attribute information.

Full-Screen FLI in practice

Let's get down to the nitty gritty. The Write I/O Approach requires three 200-byte tables, corresponding to each column. Each value on those tables correspond to each visible rasterline. For example, the first byte of each table corresponds to rasterline 50, the second byte of each table corresponds to rasterline 51, etc. The first table contains values needed for the first column of the screen, the second table contains values needed for the second column of the screen, and the third table values for the third column.

In the FLI display loop prior to the STA SCROLY command, the current rasterline is used as an index to all three tables. The values are then fetched from the tables and inserted into the code that follows the STA SCROLY command using self-modifying code techniques. When the STA SCROLY happens, the code that immediately follows it starts writing the values onto the databus, all three in a row to complete the first three columns of the screen.

There is a disadvantage with this approach. It requires that three 200-byte tables be specially constructed and stored somewhere in memory that is not mirrored by the SuperCPU. A routine would have to read in a FLI graphics file, extract information from the first three columns and store it into their respective 200-byte tables.

Pasi Ojala came up with a graph depicting the SuperCPU interacting with the VIC-II in action, showing what happens after the forced DMA retrigger at STA SCROLY. The 'LDA #$xx' would have been modified earlier in the FLI routine (before the STA SCROLY) using self-modifying code. Here is the relevant source code which takes up 4 machine cycles inside the host Commodore computer.

sta scroly : abcd, d = write Y to SCROLY on 1MHz bus CPU half - Mach. Cycle #1

```
lda #$00 : ef
sta $d022 : ghij, j = write 1 to $D022 on 1MHz bus CPU half - Mach. Cycle #2
lda #$00 : kl
sta $d022 : mnop, p = write 2 to $D022 on 1MHz bus CPU half - Mach. Cycle #3
lda #$00 : qr
sta $d022 : stuv, v = write 3 to $D022 on 1MHz bus CPU half - Mach. Cycle #4
```

There are the two shared halves consisting of a machine cycle inside the host Commodore bus, and by stashing values onto the databus, this value is carried over to the VIC-II half and is read as videomatrix data during the first three columns of the FLI screen.

```
________ = VIC-II half of the 1MHz cycle
. = SCPU synchronizes to 1MHz bus
Each char position is equivalent to a 1 (20MHz) cycle.

Mach. Cycle #1      Mach. Cycle #2      Mach. Cycle #3      Mach. Cycle #4
+-------------------+-------------------+-------------------+-------------------
__________YYYYYYYYYY___DMA____1111111111___col0___2222222222___col1___3333333333___col2___ 1MHz
abc.......ddddddddddefghi.....jjjjjjjjjjklmno.....pppppppppqrstu.....vvvvvvvvvv SCPU
```

Values on the databus which is carried over onto the VIC-II half of the databus:
DMA: DMA condition detected by VIC-II
col0: colors for column 0 read, gets the value 1 put into the bus by SCPU
col1: colors for column 1 read, gets the value 2 put into the bus by SCPU
col2: colors for column 2 read, gets the value 3 put into the bus by SCPU

An alternative approach bites the dust

The SuperCPU can also fetch values onto the databus by reading from the I/O region. If a coder were so inclined to use a 'Read I/O Approach', where is a program going to find 600 free bytes in the i/o region at $d000-$dfff? The idea is to force the SuperCPU to do a read on the databus via DMA and this can't be done with mirrored locations similar to the ones used in those VIC optimization modes. When a SuperCPU reads a value from mirrored memory, it does so from its local RAM and not the RAM that is inside the host Commodore computer. However, if the SuperCPU reads from the I/O block at $d000-$dfff, it will read a value from inside the host Commodore computer using DMA.

Unfortunately, this approach did not work when the BA line went high inside the host Commodore computer and is unworkable for a full-screen FLI mode. The SuperCPU stops for reads if BA is high, just like its 1MHz 6510/8510 counterpart.

Other Considerations.

There were some interesting observations while debugging the full-screen FLI routines. The full-screen FLI routines were originally inspired by Robin Harbron's IRQ-based IFLI routines. Because they are driven by an IRQ, the CPU is still available for normal computational tasks.

When all three videomatrix values are written to after the STA SCROLY in the line-interruptible FLI routine, the IRQ must then exit quickly with the restoration of the registers. It's a good idea to avoid writing to any mirrored location or read/write to any I/O region ($D000-$DFFF), since the SuperCPU will have to wait for VIC to finish with the data bus.

Using a raster IRQ will naturally lead to trouble, since cycle-exact timing is needed, so a CIA timer is used. The timer may be set to synchronize a PAL or NTSC machine. Then in the FLI routine the timer can be checked and indexed into a table of preset timing values so that the forced badline retrigger at STA SCROLY will always happen at the right time on the screen, no matter what the SuperCPU is doing when the VIC-II interrupted it with an raster IRQ. Thanks goes to Ninja/The-Dreams (aka Wolfram Sang) for tips on how to create a stable line-interruptible FLI routine using timers.

Source code

Without further ado, here is the source code. This source code was used in the Santa Claus FLI Demo for Wheels OS. This code will run in either Commodore 64 or 128 computers and in either PAL or NTSC systems. It did take a lot of tweaking at Points #1, #2, #3, #4, & #5 as I tried to perfect the routines as closely as possible. The full source code for the Santa Claus FLI demo can be supplied via email upon request. It is in Concept+ (geoProgrammer) format.

```
; Wedges the full-screen FLI interrupt handler in Wheels systems.
; Thanks goes to Robin Harbron for the idea of a line-interruptible FLI routine.
InstallFLI:             ; Installs the FLI routine
        jsr     ClearMouseMode  ; Turn off the mouse.
        sei
        lda     CPU DATA        ; Save 6510/8510 Location #$01.
```

```
        sta     r6510
        lda     screenMode      ; Check computer.
        bne     2$      ; Take branch in 128 mode.
        lda     #IO_IN
.byte   $2c
2$:     lda     #%00110111      ; for 128 mode only
        sta     CPU_DATA
        lda     vmcsb   ; Save original video bank info for Wheels.
        sta     oVMBmp
        lda     scroly  ; save screen Y axis
        sta     yaxis
        MoveW   $fffe, oldVector        ; Saves the old Wheels IRQ vector.
        lda     #<fli   ; Sets up fli raster.
        sta     $fffe   ; At the IRQ interrupt vector.
        lda     #>fli
        sta     $ffff
; Point #1
        lda     #$31    ; Trigger the IRQ request
        sta     raster  ; At rasterline 49.
        lda     scroly
        and     #$7f    ; Clear bit 7 of raster register.
        sta     scroly
        lda     #1
        sta     vicirq  ; Ack raster ints.
        cli
        rts

RemoveFLI:              ; Removes the FLI routine
        sei
        MoveW   oldVector, $fffe        ; Restores the old Wheels IRQ vector.
        lda     #$fb    ; Trigger the IRQ request
        sta     raster  ; At rasterline 251.
        lda     yaxis   ; restore screen Y axis
        and     #$7f    ; Clear bit 7 of raster register.
        sta     scroly
        lda     #1
        sta     vicirq  ; Ack raster ints.
        lda     oVMBmp  ; Restore original video bank info for Wheels.
        sta     vmcsb
        lda     r6510
        sta     CPU_DATA        ; Restores 6510 Port #$01
        cli
        jmp     StartMouseMode  ; Start the mouse on.
fli:    ; The actual FLI interrupt routine lies here.
        pha
.byte   $da     ;phx
.byte   $5a     ;phy
        php             ; Save processor flags.
; Point #2
        ldx     #$03    ; #$0f for PAL SuperCPU systems.
3$      dex
        bpl     3$
        lda     raster
        tax
        ldy     colOneClrs,x    ; Get colors for the first three columns.
        sty     mark4+1
        ldy     colTwoClrs,x
        sty     mark5+1
        ldy     colThreeClrs,x
        sty     mark6+1
;       ldy     backgndTable,x  ; Get background color for scanline.
;       stx     $d021
        inx
; Point #3
        cpx     #$f9    ; Have we reached scanline 249?
        bne     1$
; Point #1
        ldx     #$31    ; Restart the IRQ at rasterline 49.
1$:     stx     raster  ; By this time, the raster interrupt register is
                        ; incremented by one, and will re-trigger the
                        ; same fli routine.
                        ; This way, it is fully line-interruptible
                        ; and frees up SuperCPU time.
        ldy     #$01
        sty     vicirq  ; Ack raster ints.
        and     #$07    ; Mask out lower three bits.
        tax
        ldy     tabd018,x       ; Use preset values for vmcsb.
        lda     d011tab,x       ; Use preset values for scroly.
        sty     vmcsb   ; Select video matrix.
        sta     scroly  ; Forces the badline.
```

```
mark4:  lda #$00        ; Stores a video matrix value onto the first column.
        sta $d022
mark5:  lda #$00        ; Stores a video matrix value onto the second column.
        sta $d022
mark6:  lda #$00        ; Stores a video matrix value onto the third column.
        sta $d022
        plp             ; Restore processor flags.
.byte   $7a     ;ply
.byte   $fa     ;plx
        pla             ; Do NOT use any memory accesses to the host
        rti             ; Commodore databus in this part because it will
                        ; be blocked by the VIC-II badline.

; Point #4
tabd018:                        ; Preset video matrix values.
.byte   $78,$08,$18,$28,$38,$48,$58,$68; NTSC systems
;.byte  $08,$18,$28,$38,$48,$58,$68,$78 - PAL systems
d011tab:                        ; Preset VIC DMA retrigger values.
.byte   $38,$39,$3a,$3b,$3c,$3d,$3e,$3f

ChkAbortKey:            ; Checks the RUN/STOP keypress.
        LoadB   $dc00, #$7f     ; check for the STOP key
3$:     asl     $dc01   ; Check for the RUN/STOP keypress.
                        ; This also synchronizes the line-interruptible FLI routine.
        bcs     3$      ; Branch if it isn't pressed.
        rts
prep3Cols:              ; Prepares the first three columns of the FLI screen
                        ; Ideally, a FLI file would be loaded in and this
                        ; routine would then be called to set up the three
                        ; 200-byte tables corresponding to each column,
                        ; covering the first three columns of the screen.
        lda     #$40
        sta     mark1+2 ; Prepare the marks.
        sta     mark2+2
        sta     mark3+2
        ldy     #$00
        sty     mark1+1
        iny
        sty     mark2+1
        iny
        sty     mark3+1
        php
        sei
        lda     screenMode
        beq     1$      ; take branch in 64 mode.
        lda     $ff00
        pha             ; save 128 configuration.
        lda     #%01111110      ; select RAM at $4000
        sta     $ff00
1$:     ldy     #$00
        ldx     #$07    ; Use self-modifying code to create three
; Point #5
mark1:  lda     $4000   ; 200-byte tables for each column of the
        sta     colOneClrs+49,y ; FLI screen and each value is indexed by
mark2:  lda     $4001   ; the scanline in the FLI routine.
        sta     colTwoClrs+49,y
mark3:  lda     $4002
        sta     colThreeClrs+49,y       ; use +48 for the column offset in PAL
        clc                             ; systems.
        lda     mark1+2
        adc     #$04
        sta     mark1+2
        sta     mark2+2
        sta     mark3+2
        iny
        dex
        bpl     mark1
        sec
        lda     mark1+2
        sbc     #$20
        sta     mark1+2
        clc
        lda     mark1+1
        adc     #$28
        sta     mark1+1
        tax
        inx
        stx     mark2+1
        inx
        stx     mark3+1
        lda     mark1+2
```

```
        adc     #$00
        sta     mark1+2
        sta     mark2+2
        sta     mark3+2
        cpy     #200
        bne     mark1-2
        lda     screenMode
        beq     2$      ; take branch in 64 mode.
        pla
        sta     $ff00   ; restore 128 configuration.
2$:     plp
        rts

.ramsect $1000
; All column colors are referenced by scanline.
colOneClrs:             ; Column one colors of the FLI screen.
.block  256
colTwoClrs:             ; Column two colors of the FLI screen.
.block  256
colThreeClrs:           ; Column three colors of the FLI screen.
.block  256
```

Hopefully the full-screen FLI possibilities that the SuperCPU can now unlock will bring forth cool software for our SuperCPU's and tons of 'eye candy'.

Enjoy.

....
....
..
.

::

Main Articles

::

VIC KERNAL Disassembly Project - Part V

Richard Cini

February, 2002

Introduction

In Part 4 of this series, we discovered that of the 39 ROM routines available to be called by user code, 26 of them related to device I/O. The path to using a device from machine code was to first "open" it. So, we looked at the routines required to open and begin using a logical device.

When we concluded the discussion on the OPEN command, we left out seven routines dealing with the tape device. This is some of the most complex code I've ever seen. Unlike the IEEE serial peripherals, the tape deck is a "dumb" device, meaning that the VIC Kernal has to do the work of moving the data to/from the tape. With the serial peripherals, the Kernal just sends a character to the device that then acts on it independently of the VIC.

After opening the device, there are nine non-tape routines to deal with getting data into and out of the VIC, including talking and untalking, listening and unlistening, moving the characters in and out, and some secondary address stuff.

In this installment, we'll conclude our discussion of the OPEN procedure by looking at the tape routines that are called from OPEN. Later articles will discuss the actual movement of bits to and from the cassette and the remaining IEEE serial routines, as well as the loading and saving of files.

Tape Routines

In the last installment, we examined the beginnings of opening a device for use by a program. The reader will find that the routines are convoluted and hard to follow, with jumps and branches into apparently unrelated subroutines. Overall it appears to be ugly but highly functional code.

Here's a "calling tree" outline to the routines called by OPEN:

```
IOPEN----
        |-FIND  (analyzed in Part IV of this series)
```

```
        |-SENDSA        (ultimately handles IEEE serial stuff)
        |-SEROPN        (handles RS232 stuff)
        |
        |(all tape-related from here down)
        |-GETBFA        (get address of tape buffer)
        |-PLAYMS        (prompt user to press PLAY on tape deck)
        |   |-TPSTAT    (tape key status)
        |   |-TPSTOP    (check for STOP keyboard key during tape ops)
        |
        |-SRCHMS        (print "Searching" or "Searching for..." messages)
        |-LOSCPH        (locate a tape header with filename matching one in
        |                OPEN)
        |-LOCTPH        (locate first/next header; no filename specified)
        |   |-SETBST    (sets tape buffer start/end pointers)
        |   |-PLAYMS    (see above)
        |   |-TPCODE    (main tape code-moves bits in and out on IRQ)
        |       |-SBIDLE        (serial buss idle check)
        |       |-STOIRQ1       (put key tape vectors into table)
        |       |-NCHAR         (sets bit counter for char input operations)
        |       |-TPSTOP        (see above)
        |       |-IUDTIM        (update jiffy clock; previous installment)
        |
        |-RECDMS        (prompt user to press PLAY & RECORD on tape deck)
        |-WRTPHD        (write a tape header to tape)
        |   |-SETSBT    (see above)
        |   |-TPWRIT1   (precedes TPCODE by 12 bytes)
```

When dealing with the tape, it helps to understand that Commodore built the tape protocol with an eye towards readability under adverse conditions, including tape quality and motor speed. This made Commodore tapes probably one of the most reliable data systems when compared to TI, Apple, and Tandy, among others. Data headers and data blocks are repeated on the tape and a comparison is made between the two reads to ensure integrity. Additionally, the recording method is self-clocking so the effects of varying tape speed are minimized.

One of the first routines used in opening the tape device is determining the address of the tape buffer and making sure that it's in the $02xx page (or higher) of the system RAM. A test at IOPEN_S5 bails out with an "illegal device" error if the tape buffer isn't just so.

```
F84D    ;=============================================================
F84D    ; GETBFA - Get start of tape buffer
F84D    ;    Returns buffer address in .X (LSB) and .Y (MSB)
F84D    ;
F84D            GETBFA
F84D A6 B2              LDX TAPE1
F84F A4 B3              LDY TAPE1+1
F851 C0 02              CPY #$02                ;is buffer at $02xx?
F853 60         RTS
```

PLAYMS is called by IOPEN at IOPEN_S6. A test there determines if we're trying to read/load or write/save from/to a tape. Read mode results in the "Press Play..." message, the "Searching for..." message and two routines that search for the appropriate tape header.

```
F894    ;=============================================================

F894    ; PLAYMS - Wait for tape key on read
F894    ;
F894            PLAYMS
F894 20 AB F8           JSR TPSTAT      ;get tape key status
F897 F0 1C              BEQ TPSTEX      ;$F8B5 pressed? yes, exit.
F899
F899 A0 1B              LDY #KIM_PLAY   ;offset for "Press Play..." message

F89B            PLAYMS1
F89B 20 E6 F1           JSR MSG         ;print it

F89E            WTPLAY
F89E 20 4B F9           JSR TPSTOP      ; check for STOP key
F8A1 20 AB F8           JSR TPSTAT      ;get key status
F8A4 D0 F8              BNE WTPLAY      ;$F89E wait for PLAY switch

F8A6 A0 6A              LDY #KIM_OK     ;offset for "OK" message
F8A8 4C E6 F1           JMP MSG         ;print it and return to caller
```

This simple routine prints the "Searching for..." message only if in direct mode, and if appropriate, the filename that is being searched for. If no filename is present, the message is changed to "Searching..." (without

the "for") before it's output to the screen.

```
F647    ;=============================================================
F647    ; SRCHMS - Print "Searching for [filename]"
F647    ;
F647            SRCHMS
F647 A5 9D              LDA CMDMOD      ;direct mode?
F649 10 1E              BPL SRCHEX      ;$F669 no (programmed mode), exit
F64B
F64B A0 0C              LDY #KIM_SRCH   ;"Searching" message
F64D 20 E6 F1           JSR MSG         ;output message
F650 A5 B7              LDA FNMLEN      ;get filename length
F652 F0 15              BEQ SRCHEX      ;$F669 no filename, skip "for"
F654 A0 17              LDY #$17        ;point to "FOR" in "Searching For"
F656 20 E6 F1           JSR MSG         ;print it
F659            ;
F659            ; FLNMMS - Print filename
F659            ;
F659            FLNMMS
F659 A4 B7              LDY FNMLEN      ;get filename length
F65B F0 0C              BEQ SRCHEX      ;$F669 no filename, exit
F65D A0 00              LDY #$00
F65F            FLNMLP                  ; loop to print filename
F65F B1 BB              LDA (FNPTR),Y   ;get character
F661 20 D2 FF           JSR CHROUT      ; and print it
F664 C8                 INY
F665 C4 B7              CPY FNMLEN
F667 D0 F6              BNE FLNMLP      ;$F65F loop
F669            SRCHEX
F669 60                 RTS             ;exit
```

If the user is attempting to open a tape file with a specific filename, the IOPEN code makes a call to LOCSPH in IOPEN_S6 to find the file header associated with the filename. If the specific header is not found, the routine emits the "File not Found" error message. LOCSPH is a loop wrapper around the LOCTPH routine that searches for the "next" file header regardless of name. The secondary address parameter of the OPEN command defines whether the action is to (0) read a tape file and relocate it in memory, (1) read a tape file without relocation (a machine program), or (2) write a tape file and put both EOF an EOT markers after it. Once a header is found, the filename from the header is compared to the filename in the OPEN command to see if there's a match.

```
F867    ;=============================================================
F867    ; LOCSPH- Find specific tape header
F867    ;
F867            LOCSPH
F867 20 AF F7           JSR LOCTPH      ;search for next header
F86A B0 1D              BCS LCSPEXC+1   ;$F889 returned EOT? Go to ready

F86C A0 05              LDY #$05        ;filename offset in header
F86E 84 9F              STY TPTR2       ;save offset
F870 A0 00              LDY #$00        ;loop counter
F872 84 9E              STY TPTR1       ;save it

F874            LCSPHLP
F874 C4 B7              CPY FNMLEN      ;compare filename length
F876 F0 10              BEQ LCSPEXC     ;$F888 counter 0, exit
F878
F878 B1 BB              LDA (FNPTR),Y   ;get filename letter
F87A A4 9F              LDY TPTR2       ; offset to name in header
F87C D1 B2              CMP (TAPE1),Y   ;compare to tape header
F87E D0 E7              BNE LOCSPH      ;f867 different, get next header

F880 E6 9E              INC TPTR1       ;increment counters
F882 E6 9F              INC TPTR2
F884 A4 9E              LDY TPTR1
F886 D0 EC              BNE LCSPHLP     ;$F874 compare next character
F888            LCSPEXC
F888 18                 CLC             ; exit success
F889 60                 RTS
```

Tape searches are performed linearly, so the LOCTPH routine is used to search for the next file header on the tape starting from the current tape position. This routine is also called by LOAD through IOPEN if no filename is provided as a parameter to the LOAD (or OPEN) call. Additionally, it's called in a loop by the LOCSPH routine when searching for a specific file.

LOCTPH reads a tape block to the tape buffer using TPREAD and examines a few important fields in the file header. The fields examined indicate the file type and the filename. If the file header indicates that the file is anything other than a program file or a data file, the routine looks for another file header until it reaches the end of the tape. If BASIC is in "direct mode", LOCTPH prints the "Found" message in addition to the filename.

```
F7AF    ;=============================================================
F7AF    ; LOCTPH - Read any tape header
F7AF    ;  Header type: 1=BASIC program, 2=data block, 3=machine program,
F7AF    ;               4=data header, 5=end-of-tape marker
F7AF    ;
F7AF            LOCTPH
F7AF A5 93              LDA IOFLG2
F7B1 48                 PHA             ;save load/verify flag
F7B2 20 C0 F8           JSR TPREAD      ;read tape block to buffer
F7B5 68                 PLA
F7B6 85 93              STA IOFLG2      ;restore flag
F7B8 B0 2C              BCS LOCTPEX     ;F7E6 error, end search

F7BA A0 00              LDY #$00        ;index reg
F7BC B1 B2              LDA (TAPE1),Y   ;get header type
F7BE C9 05              CMP #$05        ;EOT?
F7C0 F0 24              BEQ LOCTPEX     ;$F7E6 yes, exit

F7C2 C9 01              CMP #$01        ;BASIC program?
F7C4 F0 08              BEQ LOCTP1      ;$F7CE yes, branch

F7C6 C9 03              CMP #$03        ;ML program?
F7C8 F0 04              BEQ LOCTP1      ;$F7CE yes, branch

F7CA C9 04              CMP #$04        ;data header?
F7CC D0 E1              BNE LOCTPH      ;must be data block-skip it

F7CE            LOCTP1                  ;program or data header comes here
F7CE AA                 TAX
F7CF 24 9D              BIT CMDMOD      ;direct mode?
F7D1 10 11              BPL LOCTPEX-2   ;$F7E4 no, continue

F7D3 A0 63              LDY #KIM_FOUN   ;setup for "Found" msg
F7D5 20 E6 F1           JSR MSG         ;print it
F7D8 A0 05              LDY #$05        ;offset to file name

F7DA            LOCLOOP                 ; loop to print filename
F7DA B1 B2              LDA (TAPE1),Y   ;print loop
F7DC 20 D2 FF           JSR CHROUT
F7DF C8                 INY
F7E0 C0 15              CPY #$15        ;21 characters max
F7E2 D0 F6              BNE LOCLOOP     ;$F7DA loop
F7E4 18                 CLC
F7E5 88                 DEY

F7E6            LOCTPEX
F7E6 60                 RTS
```

The TPREAD routine is responsible for the setup required to read or verify a block from the tape. It prompts the user to press the PLAY button on the tape deck, disables system interrupts and sets some important parameters before execution falls through to the code responsible for starting the tape IRQ routines. In many Commodore machines, tape operations are performed under the operation of a routine installed as a temporary IRQ handler -- sort of a cheap multitasking so that the system doesn't appear to be hung while tape operations are occurring. Execution ultimately comes to code at $F8EF (TPCODE) which is responsible for installing and starting the tape IRQ routine.

All of this and we haven't yet reached the bits on the tape :-)

```
F8C0    ;=============================================================
F8C0    ; TPREAD - Read tape block
F8C0    ;
F8C0            TPREAD
F8C0 A9 00              LDA #$00
F8C2 85 90              STA CSTAT       ;clear status variable...
F8C4 85 93              STA IOFLG2      ;and read/verif flag

F8C6            TPREAD1
F8C6 20 54 F8           JSR SETBST      ;set tape buffer pointers
```

```
F8C9            ;
F8C9            ; load program
F8C9            ;
F8C9            TPREAD2
F8C9 20 94 F8           JSR PLAYMS      ;wait for Play key
F8CC B0 1F              BCS TPCODE-2    ;$F8ED (in TPWRIT1)
F8CE
F8CE 78                 SEI             ;disable interrupts
F8CF A9 00              LDA #$00        ;clear work memory for IRQ routines
F8D1 85 AA              STA RIDATA
F8D3 85 B4              STA BITTS
F8D5 85 B0              STA TPCON1
F8D7 85 9E              STA TPTR1
F8D9 85 9F              STA TPTR2
F8DB 85 9C              STA BYTINF
F8DD A9 82              LDA #$82        ;Timer H constant
F8DF A2 0E              LDX #$0E        ;number for IRQ vector
F8E1 D0 11              BNE TPCODE1     ;$F8F4 (TPCODE1 in TPWRIT below)
                                        ; falls through to TPWRIT below)
```

SRCHMS also calls this small routine to determine if a key is pressed on the tape deck. TPSTAT looks at PA6 on VIA1 to determine the key state and sets up the Z flag for a compare to be performed in SRCHMS.

```
F8AB    ;=============================================================
F8AB    ; TPSTAT - Check tape key status
F8AB    ;
F8AB            TPSTAT
F8AB A9 40              LDA #%01000000  ;$40
F8AD 2C 1F 91           BIT D1ORAH      ;switch sense
F8B0 D0 03              BNE TPSTEX      ;$F8B5 not pressed, exit
F8B2 2C 1F 91           BIT D1ORAH      ;button pressed.  Setup for another
F8B5                                    ;compare later. Z=1 if pressed
F8B5            TPSTEX
F8B5 18                 CLC
F8B6 60                 RTS             ;return clear
```

One of the tests performed in IOPEN (also at IOPEN_S6) is to determine if the tape operation is a read or a write. If we're in the write mode, the code jumps to IOPEN2. At IOPEN2, the Kernal prompts for the user to press play and record on the tape deck and then writes a file header by calling WRTPHD with a control ID of $04 (a data header). Then, IOPEN writes a control byte ID of $02 (block is a data block) to the tape buffer and returns.

RECDMS is called by IOPEN to determine if a key is pressed on the tape deck and if not, sets the message flag to the "Press Play & Record" message and prints the message by calling into the PLAYMS routine. PLAYMS prints the message and then checks for the key press.

```
F8B7    ;=============================================================
F8B7    ; RECDMS - Wait for tape key on write
F8B7  ;
F8B7            RECDMS
F8B7 20 AB F8           JSR TPSTAT      ;get button status
F8BA F0 F9              BEQ TPSTEX      ;$F8B5 pressed? Yes, continue

F8BC A0 2E              LDY #KIM_RECP   ;no, remind to "Press Play & Record"
F8BE D0 DB              BNE PLAYMS1     ;exit through $F89B
```

IOPEN calls WRTPHD at IOPEN2 with $04 as the control byte (following block is a data header) to be written into the header. WRTPHD then writes some critical information into zero-page locations in advance of filling the tape buffer with the same information. At the end of the routine, the data is written to the tape in WRTPH1.

```
F7E7    ;=============================================================
F7E7    ; WRTPHD - Write tape header
F7E7    ;  On entry, .A is the header type: 2=data blk; 4=data hdr
F7E7    ;
F7E7            WRTPHD
F7E7 85 9E              STA TPTR1       ;save header type
F7E9 20 4D F8           JSR GETBFA      ;get tape buffer address
F7EC 90 5E              BCC WRTPEX      ;$F84C exit if not right

F7EE A5 C2              LDA STAL+1      ; save some program info
F7F0 48                 PHA             ;save...start H
```

```
F7F1 A5 C1              LDA STAL
F7F3 48                 PHA             ;...start L
F7F4 A5 AF              LDA EAL+1
F7F6 48                 PHA             ;...end H
F7F7 A5 AE              LDA EAL
F7F9 48                 PHA             ;...end L
F7FA A0 BF              LDY #$BF        ;buffer length-1 (191)
F7FC A9 20              LDA #$20        ; {space}

F7FE            WRTPLP1                 ; write program data to tape buffer

F7FE 91 B2              STA (TAPE1),Y   ;clear buffer
F800 88                 DEY
F801 D0 FB              BNE WRTPLP1     ;$F7FE

F803 A5 9E              LDA TPTR1       ;get header type
F805 91 B2              STA (TAPE1),Y   ;write it
F807 C8                 INY
F808 A5 C1              LDA STAL        ;get start L
F80A 91 B2              STA (TAPE1),Y   ;write it
F80C C8                 INY
F80D A5 C2              LDA STAL+1      ;get start H
F80F 91 B2              STA (TAPE1),Y   ;write it
F811 C8                 INY
F812 A5 AE              LDA EAL         ;get end L
F814 91 B2              STA (TAPE1),Y   ;write it
F816 C8                 INY
F817 A5 AF              LDA EAL+1       ;get end H
F819 91 B2              STA (TAPE1),Y   ;write it
F81B C8                 INY
F81C 84 9F              STY TPTR2       ;save buffer offset
F81E A0 00              LDY #$00        ;filename loop counter
F820 84 9E              STY TPTR1       ;save loop counter

F822            WRTPLP2                 ; write filename to buffer
F822 A4 9E              LDY TPTR1       ;get loop counter
F824 C4 B7              CPY FNMLEN      ;compare filename length
F826 F0 0C              BEQ WRTPH1      ;$F834 done

F828 B1 BB              LDA (FNPTR),Y   ;get filename char
F82A A4 9F              LDY TPTR2       ;get tape buffer pointer
F82C 91 B2              STA (TAPE1),Y   ;write char to buffer
F82E E6 9E              INC TPTR1       ;increment loop counters
F830 E6 9F              INC TPTR2
F832 D0 EE              BNE WRTPLP2     ;$F822 loop

F834            WRTPH1                  ; flush buffer to tape
F834 20 54 F8           JSR SETBST      ;set start and end address pointers
F837 A9 69              LDA #$69        ;checksum block ID
F839 85 AB              STA RIPRTY      ;save parity character
F83B 20 EA F8           JSR TPWRIT1     ;$F8EA write block
F83E A8                 TAY             ;restore start and end addresses
F83F 68                 PLA
F840 85 AE              STA EAL
F842 68                 PLA
F843 85 AF              STA EAL+1
F845 68                 PLA
F846 85 C1              STA STAL
F848 68                 PLA
F849 85 C2              STA STAL+1
F84B 98                 TYA
F84C
F84C            WRTPEX
F84C 60                 RTS             ;exit
```

SETBST is a helper routine called by LOCTPH and WRTPHD to setup the tape buffer before a tape operation takes place. It sets the starting address of the buffer to the first address of the assigned buffer range and sets the end of the buffer to start + 192 bytes.

```
F854    ;=========================================================
F854    ; SETBST - Set buffer start/end pointers
F854    ;
F854            SETBST
F854 20 4D F8           JSR GETBFA      ;get buffer address
F857 8A                 TXA
F858 85 C1              STA STAL        ;start=start of buffer
F85A 18                 CLC
F85B 69 C0              ADC #$C0        ;end=start+192
F85D 85 AE              STA EAL
```

```
F85F 98                 TYA
F860 85 C2              STA STAL+1
F862 69 00              ADC #$00
F864 85 AF              STA EAL+1
F866 60                 RTS
```

TPWRIT performs the nuts and bolts of moving cassette data in and out of the VIC. The VIC moves tape data by using a series of interrupt routines that are swapped into the IRQ vector as needed. The benefit here is that the tape IRQ code is then executed 60 times per second, along with normal processing, until the operation is complete, resulting in a cheap form of multitasking.

TPWRIT performs some setup chores before changing the IRQ vector, including clearing interrupts, ensuring that the IEEE serial bus is idle, saving the old vector, assigning the new vector and setting-up the variables used by the tape IRQ routine to actually move bits. Finally, interrupts are enabled at $F92E which starts the whole process. When the tape IRQ routine is finished it restores the IRQ vector to the standard $EABF which is detected by TPWRIT at TPCDLP2 (an I/O loop). When completed, TPWRIT exits through the TPSTOP routine. This loop also updates the jiffy clock.

```
F8E3    ;=========================================================
F8E3    ; TPWRIT - Initiate tape write
F8E3    ;
F8E3            TPWRIT
F8E3 20 54 F8           JSR SETBST      ;get buffer pointers
F8E6 A9 14              LDA #$14        ;checksum
F8E8 85 AB              STA RIPRTY      ;save it

F8EA    ;
F8EA    ; write buffer to tape
F8EA    ;
F8EA            TPWRIT1
F8EA 20 B7 F8           JSR RECDMS      ;wait for Record+Play keys
F8ED B0 68              BCS TPSTPX1     ;$F957 exit
F8EF    ;
F8EF    ; TPCODE - Common tape code
F8EF    ;
F8EF            TPCODE
F8EF 78                 SEI             ;disable interrupts
F8F0 A9 A0              LDA #%10100000  ;$A0 Timer H constant
F8F2 A2 08              LDX #%00001000  ;$08 offset for tape IRQ vector

F8F4            TPCODE1
F8F4 A0 7F              LDY #%01111111  ;$7F disable interrupts
F8F6 8C 2E 91           STY D2IER       ;save to interrupt enable reg
F8F9 8D 2E 91           STA D2IER       ;  on VIAs
F8FC 20 60 F1           JSR SBIDLE      ;wait for timeout
F8FF AD 14 03           LDA IRQVP       ;save current IRQ Vector
F902 8D 9F 02           STA TAPIRQ
F905 AD 15 03           LDA IRQVP+1
F908 8D A0 02           STA TAPIRQ+1
F90B 20 FB FC           JSR STOIRQ1     ;$FCFB .X=8 set tape IRQ vectors
F90E A9 02              LDA #$02        ;read # of blocks
F910 85 BE              STA FSBLK
F912 20 DB FB           JSR NCHAR       ;set bit counter for serial shifts
F915 AD 1C 91           LDA D1PCR
F918 29 FD              AND #%11111101  ;$FD CA2 manual low
F91A 09 0C              ORA #%00001100  ;$0C force bits 2,3 high
F91C 8D 1C 91           STA D1PCR       ;switch on tape drive
F91F 85 C0              STA CAS1        ;flag as on
F921 A2 FF              LDX #$FF        ;delay loop for high (outer)
F923 A0 FF              LDY #$FF        ;inner loop

F925            TPCDLP1
F925 88                 DEY
F926 D0 FD              BNE TPCDLP1     ;$F925

F928 CA                 DEX
F929 D0 F8              BNE TPCDLP1-2   ;$F923 outside loop
F92B 8D 29 91           STA D2TM2H
F92E 58                 CLI             ;allow tape interrupts
F92F    ;
F92F    ; wait for I/O-end
F92F    ;
F92F            TPCDLP2
F92F AD A0 02           LDA TAPIRQ+1    ;check IRQ direction
F932 CD 15 03           CMP IRQVP+1     ;standard vector?
F935 18                 CLC
```

```
F936 F0 1F              BEQ TPSTPX1     ;$F957 yes, ready

F938 20 4B F9           JSR TPSTOP      ;no, check STOP key
F93B AD 2D 91           LDA D2IFR       ;timer 1 IF clear?
F93E 29 40              AND #%01000000  ;$40
F940 F0 ED              BEQ TPCDLP2     ;$F92F continue

F942 AD 14 91           LDA D1TM1L      ; get timer 1 loword
F945 20 34 F7           JSR IUDTIM      ;update RTC
F948 4C 2F F9           JMP TPCDLP2     ;$F92F loop
```

TPSTOP is another helper routine. It scans for the keyboard STOP key, and if detected, restores the standard IRQ vector and returns to the caller's caller (the caller to TPWRIT).

```
F94B    ;=============================================================
F94B    ; TPSTOP - Check for tape stop
F94B    ;
F94B            TPSTOP
F94B 20 E1 FF           JSR STOP        ;scan STOP key
F94E 18                 CLC
F94F D0 0B              BNE TPSTPX      ;$F95C not pressed, return

F951 20 CF FC           JSR RESIRQ      ;pressed, turn drive off and restore IRQ
F954 38                 SEC             ;set error flag
F955 68                 PLA             ;pop return address
F956 68                 PLA

F957            TPSTPX1
F957 A9 00              LDA #$00        ;flag normal IRQ vector
F959 8D A0 02           STA TAPIRQ+1

F95C            TPSTPX
F95C 60                 RTS             ; return to caller's caller
```

The SBIDLE routine is used in TPWRIT to detect if the RS-232 serial bus is active and if so, will wait until it's idle before returning to continue the tape code.

```
F160    ;=============================================================
F160    ; SBIDLE - Set timer for serial bus timeout
F160    ;
F160            SBIDLE
F160 48                 PHA             ;save .A
F161 AD 1E 91           LDA D1IER       ;get IER
F164 F0 0C              BEQ SBIDLEX     ;$F172 no interrupts pending, exit

F166            SBIDLLP
F166 AD 1E 91           LDA D1IER       ;get IER
F169 29 60              AND #%01100000  ;$60 T1 & T2
F16B D0 F9              BNE SBIDLLP     ;$F166
F16D
F16D A9 10              LDA #%00010000  ;$10 kill CB1 RS232
F16F 8D 1E 91           STA D1IER

F172            SBIDLEX
F172 68                 PLA
F173 60                 RTS
```

RATS3 is at the tail end of the RAMTAS routine - the subroutine that precedes the tape vectors.

```
FCF6    ;=============================================================
FCF6    ; STOIRQ - Set IRQ vector
FCF6    ; usually called with .x=$08 or $0e. $08 points to the first
FCF6    ;       entry in TAPVEC while $0e points to the last entry
FCF6            STOIRQ
FCF6 20 CF FC           JSR RESIRQ      ;restore std IRQs
FCF9 F0 97              BEQ TPEOI       ;$FC92

FCFB            STOIRQ1
FCFB BD E9 FD           LDA RATS3,X     ;$FDE9,X set vectors from table
FCFE 8D 14 03           STA IRQVP
FD01 BD EA FD           LDA RATS3+1,X   ;$FDEA,X
FD04 8D 15 03           STA IRQVP+1
FD07 60                 RTS
```

These are the actual routines responsible for writing bits to the

tape. Various calling routines place these vectors into the IRQ vector that then gets executed 60 times per second. In order, these routines are for writing a leader block to the tape, the routine to write data to the tape, the standard IRQ vector, and the routine to read bits from the tape.

```
FDF1    ;=============================================================
FDF1    ; TAPEVC - Tape IRQ vectors
FDF1    ;
FDF1            TAPEVC
FDF1            ;       .dw $FCA8, $FC0B, $EABF, $F98E
FDF1 A8FC0BFCBFEA       .dw WRLDR2, TWRD7, IRQVEC, RDTPBT
FDF7 8EF9
```

The NCHAR routine resets the internal bit counters to their initial state before a calling routine begins to shift the bits over a serial or tape channel. It sets the bit length to 8 and resets intermediate variables to 0.

```
FBDB    ;=============================================================
FBDB    ; NCHAR - Set bit counter for new character (serial output)
FBDB    ;
FBDB            NCHAR
FBDB A9 08              LDA #$08
FBDD 85 A3               STA SBITCF
FBDF A9 00              LDA #$00
FBE1 85 A4              STA CYCLE
FBE3 85 A8              STA BITCI
FBE5 85 9B              STA TPRTY
FBE7 85 A9              STA RINONE
FBE9 60                 RTS
```

That's all of the room we have for this part of the series. All of this code and we still haven't moved the bits off the tape. So, next time we'll look at the actual routines responsible for moving bits on and off of the tape and we'll begin to look at the IEEE serial routines.

....
....
..
.

C=H 20

::

The Art Of The Minigame

Introduction

In summer 2001, an 8-bit minigame contest was held. Voter turnout may have been low, but author turnout was high, with a total of thirty entries for the C64, Spectrum, Amstrad, and Atari 8-bit. The entries and the results are available at http://demo.raww.net/minigame/, and what follows are a series of articles by several minigame authors. The articles are for enjoyment, to stimulate thought, and, hopefully, to motivate people for next year's contest. (Everyone had a great time, btw.)

Minigames -- writing tiny programs in general -- present a set of very unique challenges. Whereas many of us are used to optimizing programs for cycle-efficiency, optimizing programs for byte-efficiency turns out to be very different -- challenging, at times aggrivating, but very rewarding. I think you will find the challenges and solutions ahead to be very interesting.

The game authors were pioneers, exploring pretty new territory, and I salute all that entered the contest (especially for making such a nice C64 showing). A special mention should be made of MagerValp ("Skinny Puppy" in Swedish, in case you've been wondering), who motivated a number of lazy people (e.g. myself) to get involved with the contest.

Before diving in to the articles ahead, which contain lots of tricks to save memory, I thought it might get us in the mood to go over some general ideas and concerns for saving bytes on a C64.

It should be obvious that some balance has to be found between cycle-efficiency and memory-efficiency. Routines should be reasonably fast in most cases, but the major portion of that balance is -- has to be -- memory efficiency.

Obviously, everything that can be put into zero page should be put there,

since zero page instructions are two bytes instead of three. Equally obvious is that every list-type fetch should use the .X register, since there is no zero-page lda xx,y instruction.

Moreover, memory is initialized to various values when the computer is reset. Many zero-page locations have specific values -- by careful code design, these can be taken advantage of to avoid any initialization code. For example, many times the code needs a zero-page pointer with the low byte zero (i.e. sta (point),y). If that variable is chosen as a location which is already zero, the code doesn't have to waste four bytes on lda #00 sta point. One-time counters can also be used in this way. Finally, certain locations can be manipulated to have certain values; for example, the Tetrattack load address forces the end address+1 to be $10f0, because $10 and $f0 are the z-coordinates of two object vertices.

There is also a fairly substantial kernal to take advantage of -- routines to clear the screen, perform memory manipulations, and so on. Knowing what is available, and what the routines do, is handy.

Self-modifying code comes in handy when large portions of code can be used for similar things with just a few changes. For example, the line routine in Tetrattack uses a single routine for lines in both the x- and y-directions, by just interchanging a few INX/INY instructions.

And finally, there are things which Mark Seelye terms "injections", which seems like a good term to me. The idea is to reuse known register values whenever possible. That is, instead of having LDA #00 STA zp in a subroutine, you might put an "STX zp" in an earlier subroutine, where you know the value of .X is zero. In other words, the instruction to clear the zp location is "injected" into a different subroutine to save a couple of bytes. Every program below uses this trick in one form or another.

You'll see lots of neat tricks in the articles ahead, but the basic design framework is: put stuff in zero-page, take advantage of default zero-page values, take advantage of the kernal, always know the register values, and reuse as much code as possible.

With that, let's check out the programs.

....
....
..
.   C=H 20

==================================Part 1==================================

Postmortem: Codebreaker

by White Flame (aka David Holz)
http://white-flame.com

The most difficult part of making an entry for this compo was deciding what game to do that could fit in 512 bytes. The ubiquitous snake clones and scroll/dodge games abound; finding something different is truly a challenge. I wanted to do a Mastermind game a looong time ago, with a full-on "We're getting attacked, solve to code to save the world!" type of setting with lots of animation, but usually found myself in application and library/utility programming instead. But luckily this idea popped back into my head as something different that would fit in the 2-page limit.

The concept is pretty simple (as most 512-byte games are). There's a 4-color secret key, you get 10 guesses at it. Two numbers reflect the score for the current guess: a black number shows how many slots have the right color, and a white number shows how many correct colors are in the guess, but in the wrong slot. Experienced players should be able to deduce the correct answer in 4-6 guesses.

Random number generation

There are 3 obvious ways to get random numbers easily in the 64: SID oscillator with noise waveform, BASIC prng, or the raster counter. I decided to go for the raster solution. Obviously, you need to wait a certain amount of time between reads of the raster location, or they'd be the same or linear. To fix this, every time I wanted a random number, I'd read the raster and call a loop that many times. When it was done, $d012 should have a "random enough" number in its lower 3 bits (to get a color from 0 to 5).

```
randomDelay:
        ldx $d012
randomLoop:
```

```
        jsr delay
        dex
        bne randomLoop;
delay:
        dey
        bne delay
        rts
```

Code space saving

What's interesting is that it actually took less bytes of code and data to have custom draw loops for the 3 textual messages than setting a pointer and calling a drawText function.

```
        ldy #22
titleloop:
        lda titletext,y
        sta screenLoc-48,y
        dey
        bpl titleloop:
```

These functions assume that you're not running on an ancient kernal, so color memory is initialized to the cursor color (white).

I also had all variables (except the secret key) use screen memory, color memory, or register .Y instead of manipulating off-screen variables and then updating the screen to match:

- For entering a guess, color memory is INC'd and DEC'd, with .Y holding the current offset from the fixed screen location. JSR $e8ea is used to scroll the screen, and everything is drawn to screen line 23.

- To find out if you've already made 10 guesses, a memory location near the top of the screen was polled to see if it was still a space. As guesses are made, the screen was scrolled, and if the title line ever scrolled into that memory location, it's game over.

- The current guess score '0 0' is drawn to the screen first, then those screen values are INC'd or DEC'd as matches are found.

- Winning constitutes checking to see if the screen has a '4' in the black score location.

Score calculation

This was a bit of a pain. At first I trying to calculate the white score by the scoring rules (every matching color that was in the wrong slot), INCing the white score on every hit, then looping for exact matches, INCing the black score on every hit. This code ended up being way too big, so I got the idea to simplify the white loop: INC the white score on every color match, whether in the right spot or not, then on every black match, DEC the white score and INC the black score. This gave the same result, but with much simpler code.

Conclusion

All in all, it was an interesting challenge for me, and had I not gotten stuck in Phoenix for an extra week, I'd have finished my 2k entry. The code is pretty straightforward, and at 415 bytes total, I'm quite happy to claim having the smallest entry, even if I didn't win (why Pacman didn't beat out a low-res scroll/dodge/single-shot game is a mystery to me). Being a hobbyist C64 programmer, and my career history being more on the research end of things, having a deadline is a nice change in terms of getting a project completed. :)

Publisher: Gasman/Raww Arse
Number of full-time developers: 1 programmer
Number of contractors: 0
Estimated budget: 0 USD (0 EUR)
Development time: About 5 hours
Release date: August 22, 2001
Platforms: Commodore 64 and compatibles
Development hardware: pee sea (1.4GHz T-bird, 512MB RAM, 100GB HD, Win98SE)
3rd party tools: xa, Vice, Notepad
In-house tools: none
Project size: 290 lines of 6502 assembly code, containing 1 line of tokenized BASIC

```
--- cb.asm ---
.word $0801 ;Starting address for loader

* = $0801

.word nextLine  ;Line link
.word $0 ;Line number
.byte 158 ;SYS
.byte 50,48,54,54 ;"2066"
.byte 0
nextLine .byte 0,0 ;end of basic

screenGameOver = $0400+(4*40)+16
screenLoc = $0400+(23*40)+16
colorLoc = $d800+(23*40)+16
screen = $fd
color = $fb
peg = 81
cursor = 87
crsr_up = 145
crsr_down = 17
crsr_left = 157
crsr_right = 29

WaitStart:
jsr $ffe4
beq WaitStart

Start:

;Change to white
lda #$05
jsr $ffd2

;Clear screen
lda #147
jsr $ffd2

;Set bg colors
lda #$00
sta $d020
sta colorLoc
sta colorLoc+1
sta colorLoc+2
sta colorLoc+3
sta $ff ;for cursor
lda #11 ;dark gray
sta $d021

;Init zp pointers
lda #<screenLoc
sta screen
sta color
lda #>screenLoc
sta screen+1
lda #>colorLoc
sta color+1

;Draw text
ldy #22
titleloop:
 lda titletext,y
 sta screenLoc-48,y
 dey
 bpl titleloop:

;Randomize values

iny
newGuess
sty $02
guessAgain
 jsr randomDelay
 lda $d012
 and #$07
 cmp #$06
 bpl guessAgain
 ldy $02
 checkLoop:  ;See if the color's already in the key
 dey
 bmi goodGuess
```

```
  cmp secret,y
  beq guessAgain
 bne checkLoop

 goodGuess:  ;Save the color and put a peg on the screen to show progress
 ldy $02
 sta secret,y
 lda #peg
 sta (screen),y
 iny
 cpy #$04
 bne newGuess

Game_Loop:

;Get cursor pos
ldy $ff

;Draw cursor
lda #cursor
sta (screen),y

Key_Loop:

;Wait for key
getin: jsr $ffe4
beq getin
ldy $ff

;If up, bump color up
cmp #crsr_up
bne notUp
 lda (color),y
 and #$07
 adc #$00 ;carry is always set after the cmp
 cmp #$06
 bne colorNotHigh
  lda #$00
 colorNotHigh:
 sta (color),y
notUp:

;If down, bump color down
cmp #crsr_down
bne notDown
 lda (color),y
 and #$07
 bne colorNotLow
  lda #$06
 colorNotLow:
 sbc #$01 ;carry is always set after the cmp
 sta (color),y
notDown:

;If left,
cmp #crsr_left
bne notLeft
 ; Erase cursor
 lda #peg
 sta (screen),y
 ; Bump cursor left
 dey
 bpl cursorOKLeft
  ldy #$03
 cursorOKLeft:
 ; Draw cursor
 lda #cursor
 sta (screen),y
notLeft:

;If right,
cmp #crsr_right
bne notRight
 ; Erase cursor
 lda #peg
 sta (screen),y
 iny
 cpy #$04
 bne cursorOKRight
  ldy #$00
 cursorOKRight:
```

```
 lda #cursor
 sta (screen),y
notRight:

;Store cursor pos
sty $ff

;If not enter, goto Key_Loop
cmp #13
bne Key_Loop

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

;Erase cursor
lda #peg
sta (screen),y

;Check score

;Draw "0 0"
lda #'0
sta screenLoc+5
sta screenLoc+7
lda #0 ;black
sta colorLoc+5

;Calculate white score (any matches between the guess & key)
ldy #3
yLoop:
 ldx #3
 xLoop:
  lda (color),y
  and #$07
  cmp secret,x
  bne noWhiteMatch
   inc screenLoc+7
  noWhiteMatch
  dex
  bpl xLoop
 dey
 bpl yLoop

;Calculate black score (exact matches between the guess & key)
ldy #3
blackLoop:
 lda (color),y
 and #$07
 cmp secret,y
 bne noBlackMatch
  inc screenLoc+5
  dec screenLoc+7
 noBlackMatch:
 dey
 bpl blackLoop

;Check win (black score = "4")
lda #'4
cmp screenLoc+5
beq Win

;Check for Game Over (screen's getting full)
lda screenGameOver
cmp #32
bne Game_Over

;Scroll screen up 2 rows
jsr $e8ea
jsr $e8ea

;Copy last hand
ldy #3
copyLoop:
 lda colorLoc-80,y
 sta (color),y
 lda #peg
 sta (screen),y
 dey
 bpl copyLoop:

jmp Game_Loop
```

```
Game_Over:
jsr $e8ea
ldy #8  ;Show "YOU SUCK!"
loseloop:
 lda losetext,y
 sta (screen),y
 dey
 bpl loseloop:

jsr $e8ea
ldy #3  ;Reveal the secret key
losecopy:
 lda secret,y
 sta (color),y
 lda #peg
 sta (screen),y
 dey
 bpl losecopy
jmp WaitStart

Win:
jsr $e8ea
ldy #15  ;Show Zero Wing reference
winloop:
 lda wintext,y
 sta screenLoc-4,y
 dey
 bpl winloop
rts

randomDelay:
ldx $d012  ;Loop n times, n = current raster location
randomLoop:
 jsr delay
 dex
 bne randomLoop;

delay:
dey
bne delay
rts

wintext
.byte 1,32,23,9,14,14,5,18,32,9,19,32,25,15,21,33
losetext
.byte 25,15,21,32,19,21,3,11,33
titletext .byte 58,58,58,32,3,54,52,32,3,15,4,5,2,18,5,1,11,5,18,32,58,58,58
secret = *
--- end cb.asm ---
.......
....
..
.                                     C=H 20

==================================Part 2==================================

Tinyplay by SLJ
--------

Like many people, I was a serious Ultima guy growing up.  Not only did
I love the games (and still do), but I absolutely loved the music (and still
do).  In fact, I think the music from III and IV is some of the finest,
most musical composing ever done on the C64, and the best example of how
appropriate, atmospheric music can add volumes (so to speak) to a game.

It also turns out that I had been thinking about some new ideas for a new
music system for a while.  So when Robin said he was writing an Ultima-like
game -- well!  I'm certainly no Kenneth W. Arnold, but a set of tiny tunes
sure sounded like a neat and fun thing to do, especially in the Ultima style!
Robin removed a lot of neat features and text to add in the music, so I feel
pretty honored to have been included.

The player and tunes were written on pretty short notice; I think we started
a week or so before the deadline.  Moreover, I had to go out of town the
weekend of the deadline, getting back that Sunday, where many of the
optimizations took place in frantic coding sessions!  So it shouldn't be
surprising that the code is not as optimal as it could be -- if you get
around to looking at the source code, you'll see an awful lot of weird,
duplicate, and otherwise embarrasing lines of code which anyone with a clear
head would never have used.  But as is, the player and three tunes take up
some 428 bytes (I think), which isn't too bad.  (Later on I'll describe
```

a music compression routine which would have worked great, but that I didn't have time to implement.)

Broadly speaking, the player is a note-duration based player, using two sawtooth voices for that Ultima sound (originally, before reality kicked in, the thinking was to put some sound fx in the third voice). It uses a single play routine, using the .X register to tell which voice to play, and can play multiple tunes. The original version has several effects included like major and minor arpeggios, but memory constraints forced these to be taken out (along with the original tune, alas, alas).

The full source code for the player is at the end of this article, and can be divided into four primary parts: a one-time init routine, a routine to select a tune, a routine to play notes, and the music data. But perhaps the best way to understand the player is to begin by looking at how the data for a tune is stored.

Tune encoding

Here is the third, "castle" tune, in its entirety:

```
* Tune 3: Castle

t2v1
          dfb 97            ;gate
          dfb 24+$80-1
          dfb 70,96,72
          dfb 6+$80-1
          dfb 96,96,70,72
          dfb 12+$80-1
          dfb 74,74,96,75
          dfb 24+$80-1
          dfb 77,75,74,96,72
          dfb 6+$80-1
          dfb 96,96,70,72
          dfb 24+$80-1
          dfb 70,96,96,96
          dfb 00
t2v2
          dfb 97
          dfb 24+$80-1
          dfb 34,41,46,41
          dfb 34,41,46,41
          dfb 58,62,53,57
          dfb 58,53,46,96
          dfb 0

end
```

The first chunk of data (t2v1) is the voice 1 data, and the second chunk the voice 2 data (t2v2).

There are 12 notes in an octave, and eight octaves total, for 96 possible notes. In the player, data bytes 0-95 represent these notes -- the player just uses the data as an index into a frequency table. This leaves bytes 96 and above free for other uses; also, since the lower note values are rarely used, values like 0, 1, etc. are also free in principle.

Byte values >= 128 are used to specify the note duration. When the player encounters these byte values it simply strips the high bit and stores the result. When a new note is read in, this stored duration is placed into a counter which is decremented on each player call.

In the castle tune above, the duration for voice 2 is only set once, at the very beginning -- every note has the same duration. In the voice 1 data it is set several times, to change the duration when appropriate.

For some reason the code decrements the duration counter until it becomes negative, instead of decrementing it until it becomes zero. I'm sure it was because of something relevant in an earlier version of the player, I just totally don't remember what. But that is why the durations above are encoded as 24+$80-1 -- the -1 is to make it go negative, instead of to zero. Looks weird.

Byte values >95 and <128 are available for 'special' features. The player supports one "effect": the gate toggle. The player can either leave the gate bit on all the time or can turn it off right as a note is about to finish

(say, when the duration counter decrements down to 2). A byte value of 97 turns the gate toggle on -- in the castle tune above it is the first byte of data. In other tunes, a value of 100 is used to turn off gate toggling. (Historically speaking, values 98 and 99 were used for major and minor arpeggios, and 100 was for no fx at all.)

The byte value 96 is used as a "rest" (or hold) -- the player resets the duration counter but does not reset the gate bit or the note value. When there is no gate toggling this provides a way of holding a note for longer than the basic duration. When there is gate toggling, it starts the release cycle, and lets notes fade away. In the castle tune above, the first two notes in voice 1 are "70 96" -- a note, then a rest, during which the note fades away.

Finally, the value 0 is used to indicate the end of data. While 0 is technically a note it never gets used as such. When the player hits a zero it simply resets the data pointer.

So with all that in mind, let's take a look at the player.

The Player

Before playing a tune, a one-time routine is called which sets up the frequency tables; when the player reads a new note, it looks up the SID frequency settings in this table, along the lines of:

```
        ldy note,x
        lda freqlo,y
        sta $d400
        lda freqhi,y
        sta $d401
```

Frequencies in different octaves are related by powers of 2. When you go up an octave, the frequency doubles -- for example, the frequency of C-5 is twice the frequency of C-4. The routine starts with a table of twelve frequency values for the highest octaves, and copies those values into the end of the frequency table (e.g. freqlo+95, freqlo+94, ..., freqlo+84). After copying, it divides each value by 2 and repeats the process -- now the frequencies correspond to the next lower octave -- and this is done until the frequency table is full. Piece of cake.

Once the frequency table is set up, the main play routine is called with the tune number in .A. If the tune number is different than the current tune, the player simply resets the music data pointers and note durations, and falls through to the main player routine.

And the main player routine simply decrements the duration, and when it becomes negative reads in the next data.

Because the process -- decrement duration, read next data, store frequencies in SID -- is the same for each voice, a single routine is used for both voices one and two, using .X as the voice index. That is, instead of

```
        dec duration
```

the code can use

```
        dec duration,x
```

and instead of

```
        sta $d400
```

the code can use something like

```
        ldy SidOffset,x
        sta $d400,x
```

where .X is 0 or 1.

And that's about all there is to it -- the code is pretty straightforward.

Compressing Music

Most music has the property that it doesn't jump around all over the place, but rather notes progress in relatively small intervals (seconds, thirds, fifths, etc.). For example, a major chord (as used in a typical arpeggio) is

```
        note  note+4  note+7  note
```

So the idea is to use a differential scheme, encoding the note _intervals_ instead of the note _values_. That is, the above could instead be encoded as

```
        x  4  3  -7
```

where each value is added to the current note value. Thus, if intervals are restricted to -8..7 then only four bits are required, cutting the data size in half. Well, not quite of course, because an escape code is needed to specify notes outside of the -8..7 range, not to mention durations and fx settings. Luckily, not all interval values are used, so these can be used for escape codes; code 96 -- the rest or "hold" note -- can also use an otherwise unused value.

So the decompression code looks something like

```
        read next four bits
        if escape code then read next byte
        otherwise add to current note value
```

Pretty simple, and leads to quite substantial savings. Ah, but for a little more time...

Another possibility arises if not all note values are used. I set up the frequency tables to contain all 12 notes in eight octaves. If not all twelve notes are actually used -- if none of the tunes contain a D# or whatever -- then a few bytes can be shaved off of the frequency table. But this means a lot of renumbering and such... it all depends on just how much you want those extra bytes!

And that's about all there is to it.

Source code

```
*
* Tiny music player, for the
* minigame contest.
*
* slj 9/23/01
* sjudd@ffd2.com
*

                        ;         org $2800-491
         org $2800-425

ARPDEL   = 2
SR       = $c9
GATEDEL  = 2

curtune  = $fe
curdur   = $fc

tunepos  = $b0            ;position in note sequence
seqpos   = $b2            ;position in seq list
dur      = $b4            ;2 bytes each
notefx   = $b6
curnote  = $b8
newnote  = $ba

arpdur   = $bc
arpoff   = $bd

noteptr  = $fa
temp     = $fa
freqtab  = $c000




*
* Code begins here
*
start

*--------------------------------
         do 0
         jmp InitFreq
```

```
         jmp Play
         jmp Reset
         fin
*------------------------------

*
* InitFreq -- set up note table
*
InitFreq
         ldy #192
:l2      ldx #24
:loop    lda freq-1,x
         sta freqtab-1,y
         dex
         dey
         lda freq-1,x
         sta freqtab-1,y
         lsr freq,x
         ror freq-1,x
         dey
         beq :xit
         dex
         bne :loop
         beq :l2
:xit     rts

*
* Play routine
* tune in .A
*

Reset                    ; Force a tune reset
         sec
         ror curtune
                         ; ldx #255
                         ; stx curtune
Play
         ldx #1
         cmp curtune
         beq PlayTune
         sta curtune

         ldy #00
         sty tunepos
                         ;         sty tunepos+1
         sty dur
         sty dur+1

         lda #15
         sta $d418
         lda #SR
         sta $d406
         sta $d406+7


*
* Main routine
*
* voice in .X (0, 1)
*

SetTune
         ldy curtune
         txa
         beq :v1
:v2      lda v2tunepos,y
         bne :sta
:v1      lda v1tunepos,y
:sta     sta tunepos,x


PlayTune

Get
         lda #00
         sta newnote,x
         dec dur,x
         bpl DoFx

:next    ldy tunepos,x
         inc tunepos,x
```

```
         lda tunes,y
         beq SetTune
         bpl :c1
         and #$7f          ;$80+dur
         sta curdur,x
         bne :next
:c1      cmp #96
         bcc :setfreq
         beq :hold
         sta notefx,x
         bne :next
:note                      ;sta curnote,x
                           ;         sta newnote,x
                           ;         jsr setfreq
:SetFreq
         asl
         tay
         lda Freqtab,y
         pha
         lda Freqtab+1,y
         ldy SidOffset,x
         sta $d401,y
         pla
         sta $d400,y
:gate
         lda #$21
         ldy SidOffset,x
         sta $d404,y


:hold    lda curdur,x
         sta dur,x



DoFx
         ldy notefx,x
         cpy #97
         bne AllDone

         ldy dur,x
         cpy #GATEDEL
         bcs :c1
         lda #$20
         ldy SidOffset,x
         sta $d404,y
:c1

AllDone
         dex
         bpl PlayTune
RTS      rts


*
* Frequencies
*
freq
         da 34334
         da 36377
         da 38539
         da 40831
         da 43258
         da 45831
         da 48557
         da 51444
         da 54502
         da 57743
         da 61177
         da 64815

                           ;arptab dfb 0,4,7

*
* Tune positions (offset into sequence
* list)
*
v1tunepos
         dfb t0v1-tunes
         dfb t1v1-tunes
         dfb t2v1-tunes
```

```
v2tunepos
          dfb t0v2-tunes
          dfb t1v2-tunes
          dfb t2v2-tunes


tunes
                          ; dfb 00           ;dummy byte
*
* Sid offset table
*
SidOffset
          dfb 00
          dfb 07


t0v1
          dfb 97
          dfb 10+$80-1     ;dur=10
          dfb 42,47,51
          dfb 42,47,51
          dfb 42,47,47,47
          dfb 51,51,51
          dfb 54,52,51
          dfb 52,51,49
          dfb 51,51,51
          dfb 49,49,49
          dfb 51,49,47
          dfb 49,47,46
          dfb 44,44,44
          dfb 46,46,46
          dfb 47,51,47
          dfb 46,42,46
          dfb 47,96,96
          dfb 96,96
          dfb 00

t0v2                      ;v2
          dfb 97
          dfb 10+$80-1
          dfb 35,35,96
          dfb 35,35,96
          dfb 35,35,96
          dfb 35,39,96
          dfb 39,42,96
          dfb 42,30,96
          dfb 30,35,96
          dfb 35,34,96
          dfb 34,32,96
          dfb 32,32,96
          dfb 32,32,96
          dfb 32,34,96
          dfb 34,35,96
          dfb 35,30,96
          dfb 35,35,96
          dfb 35,35,96
          dfb 00

* Tune 2: Combat

t1v1
          dfb 100          ;No gate
          dfb 11+$80-1     ;dur=11
          dfb 51,54,58,59,96
          dfb 58,57,54,51,54,58,63,96
          dfb 62,61,60,59,56,53
          dfb 58,54,51,56,53
          dfb 46,50,53,56
          dfb 59,58,57,54
          dfb 0

t1v2                      ;v2
          dfb 100
          dfb 11+$80-1
          dfb 15,27,15,39
          dfb 15,27,15,39
          dfb 15,27,15,39
          dfb 15,27,15,39
          dfb 47,44,41,51,46,42,47,44
          dfb 22,34
          dfb 22,34
```

```
         dfb 22,34
         dfb 22,34
         dfb 0

* Tune 3: Castle

t2v1
         dfb 97           ;gate
         dfb 24+$80-1
         dfb 70,96,72
         dfb 6+$80-1
         dfb 96,96,70,72
         dfb 12+$80-1
         dfb 74,74,96,75
         dfb 24+$80-1
         dfb 77,75,74,96,72
         dfb 6+$80-1
         dfb 96,96,70,72
         dfb 24+$80-1
         dfb 70,96,96,96
         dfb 00
t2v2
         dfb 97
         dfb 24+$80-1
         dfb 34,41,46,41
         dfb 34,41,46,41
         dfb 58,62,53,57
         dfb 58,53,46,96
         dfb 0

end
```

....
....
..
.

==========Part 3==========

MagerTris -- by Per Olofsson <magervalp@cling.gu.se>

Way back in 1990 or thereabouts I wrote a Tetris clone in basic on the Amiga. When the compo was announced I scratched my head for a bit and figured that I should have a good chance of fitting Tetris in 512 bytes. If I couldn't I would at least have a good engine for the 2K compo.

The Game

This puzzle game was invented in the 80s by the Russian programmer Alexey Pazhitnov. The game has a vertical field where one of seven puzzle pieces appears at the top and falls towards the bottom. The player can rotate the piece 90 degrees or move it left or right. When it reaches the bottom the game field is checked for filled lines that are removed. The player is awarded points for removing lines (with a bonus for clearing more than one in one go) and the game is over when the screen fills up so that new pieces can't enter the game field.

Drawing the Screen

The screen is filled with inverted space and the game pieces and the border is drawn into the color ram. There aren't any real tricks used here, the only one is that the color of the border is a side effect of the last character printed, ascii 13 (light green).

Random Numbers

We need a decent source of random numbers to generate a random Tetris piece. The SID chip's noise waveform is probably the best one available on the C64, and fortunately the code to access it is really short. To initialize I used:

```
initrnd         subroutine
                lda #$81        ; enable noise
                sta $d412       ; voice 3 control register
                sta $d40f       ; $81xx as the frequency
```

and to get an 8-bit random number all you have to do is lda $d41b.

The Tetris Pieces

Tetris has 7 puzzle pieces that you have to store in a table. As every piece consists of four boxes we need a total of 7*4 = 28 entries. The smallest tetris table is probably the one where each piece is represented by 8 bits, like this:

```
        0010    0000    0011    0110    0100    0010    0110
        0111    1111    0110    0011    0111    1110    0110
```

The table is very compact, a mere 7 bytes, but then you need code to rotate every piece and accessing the table is somewhat clunky on the 6502. I've seen an x86 version that did this though, with a total size of less then 256 bytes. However, on the 6502 I needed four bytes for each piece for the code to be reasonably compact and I also kept all the rotated pieces in the table, for a total of 112 bytes. Fortunately, as every piece has a center box we only need to store 3 which trims the table to 84 bytes. But what do we actually store? As the screen is painted to color ram I used a pointer to the current position. In the table I simply stored the offset in color ram from the center box. With indirect indexed addressing the routine is nice and short.

Failure

The rest of the program is fairly straight forward. User input is just a cmp loop, there are dec zp timers for most events, and a raster wait to time everything. This is also where I failed to fit everything inside the 512 byte limit, as everything put together took about 600 bytes, even after some serious optimizations. There are two very similar routines: the one that draws a piece on the screen and the one that checks for collisions. Both iterate through the four boxes in a piece, but because of the way I used some zeropage pointers I couldn't merge the two routines into one without rewriting most of the code.

Success

As rewriting everything meant too much work, I just decided to go for plan B: write a 2K entry. With a 600 byte engine there was plenty of space to add features, and at around 700 bytes or so compressíon starts to make sense -- pucrunch breaks even around there somewhere, and the final binary is actually 6500 bytes uncompressed. I could easily fit a title screen complete with custom graphics, a nice tile set for the tetris pieces, and even a hiscore list that saves to disk. The only trick I used here was that the custom character set was EOR:d with the ROM font, and as most characters are the same or very similar it compresses much better that way.

And that's the story of the 2K MagerTris.

....
....
..
.

C=H 20

==========Part 4==========

Tuff -- Compressing tiny programs

----------> Stephen L. Judd <sjudd@ffd2.com>

Most compression schemes address the problem of taking a large program or data file and compressing it down. But have you given much thought to the problem of trying to compress a _small_ program, even a 512-byte program?

I spent some time trying to come up with a compression scheme for tiny programs -- saving even 2 or 3 bytes in tetrattack would have been helpful. The effort was basically a failure, as the best I thought I could do was to break even (on paper, for tetrattack). I never actually implemented the code either -- just worked it out on paper.

It is fairly interesting though, and maybe these preliminary efforts will give other people ideas (especially for compressing 2k programs). And after writing this article, I now think it might have really worked, and saved a handful of bytes -- maybe next year?

Tiny Compression Basics

If you remember your C=Hacking #16, there are two basic approaches to compression: taking a fixed-length input while giving a variable-length output, or taking a variable-length input and producing a fixed-length output. An example of the latter is LZ compression -- looking for repeated strings of length 2, 3, etc. and replacing those with a single byte (or two bytes, or whatever). An example of the former is a Huffman tree, which

replaces each byte with a string of bits, using fewer bits for the most common bytes.

To decompress the compressed data, of course, a decompression routine is needed. For tiny compression, this routine obviously needs to be as tiny as possible. Most C64 decompression programs copy themselves somewhere before decompressing the data stream to wherever it is supposed to go. Obviously, if we put a few restrictions on the data to be decompressed the decompression routine can be made smaller, and more specific to the task.

Finally, it is worth mentioning that a tiny program in general does not use all possible byte values.

Now, first consider Huffman encoding. In a typical Huffman decoding scheme, the decoder reads a bit at a time, traversing the Huffman tree with each bit, until a byte is hit. This implies a fairly complicated decoder, and worse the tree must be stored. Finally, the tree can be fairly large, if the number of possible symbols is large. So Huffman doesn't seem like a good approach.

So, consider LZ-style encoding: replacing n-byte sequences with a shorter code. In LZ, the n-byte code is replaced with a reference to the previously decompressed bytes -- a code which says how far back to go, and how many bytes to copy. What a drag -- an escape code, the distance backwards, and the number of bytes to copy. For a byte-aligned decompressor, this implies at least two bytes (since the decompression routine must be small, a byte-aligned decompressor is of greater interest).

So far so bad. But, since a tiny program doesn't use all possible byte values, perhaps a simple substitution method is possible -- that is, replacing various 2-, 3-, 4- etc. byte sequences with one of those bytes. Alas, the numbers don't work out very well: in the above scheme, let's say an n-byte sequence appears m times, for a total of n*m bytes. We replace each of those sequences with a single byte, giving a savings of

```
n*m - m
```

bytes. But we also need to store the original sequence in a table somewhere, plus the byte corresponding to that sequence. This means another n+1 bytes, so the total savings is

```
n*m - (m+n+1)
```

So, for example, if we have a 2-byte sequence repeated 3 times, the savings is

```
2*3 - (2+3+1) = 0 bytes!
```

The issue is that replacing a 2-byte sequence with a 1-byte sequence saves one byte, and this must happen at least three times to overcome the three bytes of table storage. With 4 repetitions, one byte is saved. And so on. The net result is that you have to have a _lot_ of sequences repeated a lot of times to get any savings, and somehow this savings needs to be greater than the decompression code.

Note that, in general, if there are only a few sequences to replace then some custom code can be shorter, but as the number of sequences increases then generic code, with sequences stored in tables, quickly becomes shorter.

The final dagger in the heart of these schemes is the program itself. I wrote a simple program to find all the repeated strings in tetrattack. There were numerous 2-byte sequences, but rarely repeated more than three times. There were a few 3-byte sequences, and even one 4-byte sequence. But overall, there just aren't enough sequences, especially long sequences, to make any sequence-shortening scheme worthwhile.

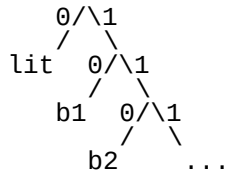
There are, however, lots of repeated bytes. For example, there are an awful lot of STA instructions and zp variables, and usually a lot of zeros. This suggests taking another look at a Huffman-style scheme to replace those bytes with a smaller number of bits.

One major problem identified earlier is storing the Huffman tree -- it's just too big. So, the first thing to consider is to just compress the most common bytes -- maybe the top five or eight bytes or whatever -- to make the tree smaller. But that still leaves a substantial decompression code to traverse the tree, and a tree with several empty nodes in it.

Once again, if there are only a few nodes then custom code might be shorter, but beyond a few nodes it's more practical to store the tree and have some

generic traversal code.

But consider the following alternative to a Huffman tree: what if we had a tree which looked like



where "lit" is a literal byte and b1, b2, etc. are encoded bytes. This would encode the most common byte with two bits (10), the next-most common byte with three bits (110), and so on, and encode literal bytes with nine bits (0byte).

To see if this is a viable scheme, check out the numbers. Let's say that we encoded just the top three most common bytes in a program; in an early version of tetratack those bytes were

```
00      $2F occurances
03      $1A occurances (ZP variable)
8D      $18 occurances (STA opcode)
```

In a real implementation, I'd choose zp variables to be common opcodes like $8D, which would double the number of occurances of that byte, but let's just consider the above numbers in a 512-byte program. Those top three bytes total $5A bytes, almost 1/8 of the total bytes (the top seven bytes take a little less than 1/3 of the total, by contrast). The number of bits required is

```
9*($200-$5A) + 2*$2F + 3*$1A + 4*$18 = $FE2 bits = $1FD bytes.
```

so, a whopping 3 byte savings -- just enough to store the three "tree" values. For the top seven bytes, a similar calculation suggests $01D8 bytes of compressed data, again for code not optimized for compression. For a more optimized code, with zp variables the same as common opcodes and such, the total bytes could conceivably be on the order of $1C0 bytes -- maybe 64 bytes of savings (but again, I never actually tried coding a more optimized version so I don't know for sure).

Of course, we haven't talked about decompression yet. One important feature of this tree is that the "treeness" of it is unimportant -- it's really just a substitution code. Remember that the data is encoded as

```
0+8 bits        literal byte
10              byte 1
110             byte 2
1110            byte 3
```

and so on. A decompression algorithm simply reads bits until a 0 is found, and then looks up the relevant value in a table. The exception is if the first bit read is 0, in which case the code needs to read the next eight bits and output the byte. There is no tree traversal, in the sense of looking up nodes and moving left/right, to speak of.

Reading bits is trickier than reading bytes. After every eight bits read some sort of memory pointer has to be incremented to the next eight bits. Moreover, when reading codes like 110 or 1110 the number of 1's read has to be counted too, to be used as an index into the lookup table. And if there's a literal byte then the next eight bits need to be read, regardless of their value.

When reading the bitstream, after every eight bits the pointer into that bitstream must be incremented. To count eight "global" bits at a time, I use a zp variable "count" and just rotate a bit through the different bit positions. Every time it turns zero, the data stream pointer is incremented. The advantage of this method is that it frees up a register (and winds up saving a byte overall).

To count the number of "local" bits read, I use the .Y register. Initially it is set to zero, and increments as 1s are read. For a literal byte, I set .Y to -8, and put a little check in the bit reader for negative .Y; that way, it simply increments to .Y=0.

When bit=0 is finally found, the code simply does an lda data,y to get the coded data. Since .Y counts the number of bits read, .Y will be at least 1 for any coded data, but .Y=0 for a literal byte as described above. Therefore, for a literal byte, the code can simply rotate bits into the

location "data", and use the same lda data,y instruction to fetch the
literal byte.

With all that in mind, here's the code I finally came up with (on paper!):

```
Tuff+1
[data lookup table]

Source  [compressed data bitstream]

:lit    ldy #-8
:getbit lsr count       ;Increment pointer every eight bits
        bne :skip
        ror count
        inx
        bne :skip
        inc :src+2
:skip
:src    asl Source,x    ;Get next bit
        bcs :next
        tya             ;.y=0 means first bit is 0
        beq :lit        ;so read next 8 bits
        bpl :found      ;If .y<0 then literal
:next   rol Tuff        ;Save bit
        iny
        bne :getbit     ;until .y=0
:found
        lda Tuff,y      ;Fetch code/literal byte
        ldy #00
        sta (zp),y      ;Output data
        inc zp
        bne :getbit
        inc zp+1
        bpl :getbit
[Uncompressed code -- (zp) points here!]
```

for a total of 45 bytes, plus 7 bytes for the "Tuff" byte table -- total
of 52 bytes, compared with a possible 64 byte savings (maybe I should have
tried to implement this after all!). Hopefully someone can improve on
this further. (And "Tuff", btw, is a contraction of "TinyHuff").

Now, there may be a few raised eyebrows here which I hope I'll lower. This
code makes several assumptions. It can either be entered in at ":src",
or at ":getbit" depending on the value of "count", the global bit counter.
I assume count is already initialized (e.g. default kernal/restart value),
either to 1 (in which case :getbit is the entry point) or $80, in which case
:src is the entry point, with the "Source" stream modified accordingly in
each case.

I further assume that .axy have their default values of 0 when the routine
is called from BASIC.

I also assume "zp" is initialized to the correct value already. One easy
way to do this is to use $ae/$af, the end load address+1, but other
options are certainly available.

Finally, you may have noticed that the thing just keeps outputting data
until zp=$8000 -- correct. Unless you really really care about garbage in
memory above the program, there's no reason to waste bytes on an "end of data"
check.

Another variant

I spent a little time looking at a variant of this scheme: using a fixed
length bit sequence, for example, assigning four bits to each substitution
byte, i.e.

```
        0+      literal byte
        1000    byte 1
        1001    byte 2
        1010    byte 3
        ...
        1111    byte 8
```

The numbers work out pretty well, compression-wise, but at the time I felt
the decompression code was more complicated and that it wouldn't break
even. Looking at it now, though, I'm starting to think otherwise -- that
loading three bits at a time isn't any more complicated that fetching
eight bits so code can be reused, and there isn't any issue about checking
for a 1 bit, so the decompression code ought to be even simpler. So this

may actually be the better scheme.

Sigh... I don't know about you, but every time I revisit a problem I wind up feeling much stupider.

Final thoughts

So, that's it: assign shorter bit codes to the most common bytes, and use a highly optimized decompression routine. This method is very specialized, obviously, but might, just might, make it possible to save a few bytes on a tiny program that has been optimized for compression -- using zp locations the same value as common opcodes, reusing vars as often as possible, and so forth.

The reason this works is that, in general, repeated long strings in a small program just don't happen, whereas repeated bytes are common and make up a substantial fraction of the total.

Pelle (magervalp@cling.gu.se) has suggested the possibility using the first eight bytes of the compressed program for the 'Tuffman' table -- to structure the code to make these common bytes, and thereby save eight bytes by not having to store the 'tree'. Might save a few more bytes with some programs.

Finally, while writing this article, I started to wonder about byte sequences with "holes", for example, <byte skip byte skip> for things like STA zp1 STA zp2. I wonder if there's some way of taking advantage of these repeated "mask" patterns.

But I leave that thought for another person, for another day.
....
....
..
.   C=H 20

==========Part 5==========

Tinyrinth
by Mark Seelye (aka Burning Horizon)
mseelye@yahoo.com
http://www,burninghorizon.com

Introduction
    When the mini-game competition was first announced I was originally not interested because of time considerations -- I spend most of my time at work, or watching the kids. But once I saw that nearly everyone on EFNet #c-64 was doing one, I thought it would be fun if I could just keep it simple.

    I had no idea what game to do. I had a few other ideas but I settled on a maze game because I thought it'd be an interesting challenge doing a maze algorithm and have game code in under 512 bytes. As it turned out, I ended up getting the code down to like 475 bytes, but the version I turned in was 509 bytes. (I ended up shaving an additional 34 bytes just before the due date.)

    The other reason for the maze game is I have this weird thing I do where in each language I know I code this little maze generation algorithm that I made a long time ago. I had used the algorithm in ASP, VB, C, and a few others but for some reason I had never done it in 6502! Imagine that! It was going to be a bit of a challenge because I had never done it in so few instructions, and never in ASM. It would have to lose some weight and fluff - but I knew I could do it. So obviously there are MANY better ways to generate a maze, but in light of trying to make it small this routine ended up being such a strange mutation of my idea that I'm surprised it worked at all.

Setup
    Before I do anything I must setup some things. This is also used when a player dies OR enters a new level. The only difference between a reset of the game and moving to the next level is the ldx #$00: sta $53. The $53 is also used to set the color of the maze so it rotates from a level one of white, to a level 16 of BLACK (which is evil because then you can't see the maze except for the small hint of the corners on a turn.)

```
; Initialization
setup = *
gameover = *
        ;Setup game variables
        ldx #$00
        stx $53         ; Start at level 1 (to be inc'd)
init = *
        ;Setup level
```

```
        inc $53         ; next level
        ldx $53
        stx $54         ;Counter for drawing Keys (next level)
        stx $0286       ;Character Color to use on clear (e544)

        ;Set Render Cursor Start Pos / Player Color
        lda #$05
        sta EBCM1       ;Set ebcm color PLAYER to GREEN ($d022)
        sta $45         ; Cursor/Player Position X (0-39)
        sta $46         ; Cursor/Player Position Y (0-24)
        ;Clear Screen
        jsr $E544       ;clear screen set to char color ($0286)

        lda #$5b
        sta $d011      ;turn on EBCM
        lda #$18
        sta $d018      ;Activate Cset
```

As you can see that I mention extended color background mode, I'll talk about that later. You also might notice I use a character set, but that I will talk about a little later as well, first I want to jump right into the maze generation.

Maze Algorithm

The easiest way to describe how the algorithm works is to compare it to a worm. The worm eats through an area until it gets blocked by one of its former meals. While eating the worm is counting how many meals he has, once he gets to a certain number he knows he can eat no more.

With that said, I'm sure you're completely confused and probably think (know?) I'm crazy. So now for the technical explanation. Starting with a 2-dimensional grid of some size, initialized with 0's, set a start point for the formation of the maze within the grid somewhere. After that has been taken care of the rest follows simple logic:

```
        Have we reached every part of the grid?
            if so: done;
            if not: Can we grow the maze into an adjacent area from the
                    current position?
                if so: Grow in one of those directions at random, count it,
                       continue;
                if not: Find a place in the maze that can grow, continue
```

I had to change the order of the logic around to get it to better fit into a smaller number of bytes. So I moved the "find a place to grow from" routine, and the "Have we reached every part of the grid" check after the check if we can grow in any direction from the current position.

To check in all directions from the current position I used a subroutine. The subroutine has 5 entry points which translate to: cangrow up/right/down/ left, and the 5th entry point is actually the first entry point - it checks to see what is in .a and checks the direction that value represents. To check the direction I use a kernal routine to set some zp so I can easily check the value on the screen to the left, right, below, or above the current position. This is also where the boundaries are set: 40 columns wide and 25 characters high.

```
popgridloop = *
        ldy $45         ;xpos
        ldx $46         ;ypos
cangrowxy = *           ;Can grow in any direction?
        jsr cgup        ;check up
        beq _cgxy       ;if 0 then we can grow
        inx             ;offset up check
        jsr cgright     ;check right
        beq _cgxy       ;if 0 then we can grow
        dey             ;offset right check
        jsr cgdown      ;check down
        beq _cgxy       ;if 0 then we can grow
        dex             ;offset down check
        jsr cgleft      ;check left
_cgxy   beq growloop    ;if 0 then we can grow
```

....The subroutine!... There is a weird "injection" in the bottom of this routine to save a couple bytes.. it is explained in the comments to the code.

```
;Check if a cell can grow a direction
;1-up 2-right 3-down 4-left
; (y xpos, x ypos, a=dir)  x/y switched for indirect lda($xx),y below
; return: a == 0 : true (can move)
;         a <> 0 : false (cannot move)
```

```
cangrow = *
    cmp #$01
    beq cgup
    cmp #$02
    beq cgright
    cmp #$03
    beq cgdown
    ;cmp #$04
    ;beq cgleft *** not needed falling through
cgleft = *
    dey         ;set xpos - 1
    cpy #$ff    ;check xpos
    beq cgno
    bne cgloadcell
cgright = *
    iny         ;set xpos + 1
    cpy #$28    ;check xpos (<40)
    beq cgno
    bne cgloadcell
cgup = *
    dex         ;set ypos - 1
    cpx #$ff    ;check xpos
    beq cgno
    bne cgloadcell
cgdown = *
    inx         ;set ypos + 1
    cpx #$19    ;check ypos (<25)
    beq cgno
    ;*** fallthrough, bne cgloadcell not needed
cgloadcell = *
    lda #$1f
loadcell = *        ;x = ypos, y = xpos, a = and value
    sta $50
    jsr $e9f0   ; * sets $d1 $d2
    lda ($d1),y     ;load byte  (x pos in y reg!)
cgno = *
    and $50  ;#$1f = use only low 5 bits!
    ;rts see below!
;  This is mixed with the rts because the first byte would
;  be wasted
growfrom = *
    rts
    .byte 1
; this is part growfrom part growto  48 is used for both
;  again first byte would be wasted so we overlap with the previous
growinto = *
    .byte 2, 4, 8, 1, 2

;explanation of above
; 0  1  2  4  8
;       0  4  8  1  2
;rts 1  2  4  8
;          4  8  1  2

*From Mapping: ;59888         $E9F0
               ;Set Pointer to Screen Address of Start of Line
               ;This subroutine puts the address of the first byte of the
               ;screen line designated by the .X register into locations
               ;209-210 ($D1-$D2).
```

Well, if the code at "cangrowxy" discovers that it cannot grow in any direction from the current position, it has to find a place to grow from. To do this it falls into the "findgrow" routine; the findgrow routine is interesting because it is a "stateful" routine, meaning that upon reentry it resumes what it was doing before. The reason for this is I didn't want it restarting at the top of the maze each time it entered this find routine - I wanted it to continue to search from where it left off. This routine works by "walking" the current x and y positions through the grid. To walk it through it keeps track of where it left off and it just keeps setting the x and y pointers to the next place and "returns" to the start of the generation portion of the code.

The other advantage to doing it this way is I can reset this routine to place a "key" at a dead end in the maze. This reset is done elsewhere in the maze by setting a zp value to 0. When the reset is done the findgrow routine runs a small piece of code to reset itself, but before it does this it places a key if it has not exceeded the max number of keys.

Here is the find grow routine, each line of code is documented to hint at what it is doing:

```
; *** fall into findgrow
```

```
findgrow = *
        lda $4b         ; Check byte 0 != resume findgrow
        bne _fgresume
        sta $49         ;Reset Findgrow Xpos
        sta $4a         ;Reset Findgrow Ypos
        inc $4b         ;Set findgrow flag to resume (<>0)

        ;Place keys in corners (injected here for ease of placement, d1/d2 is
        ;pointed at a dead end)
        iny             ; offset left check
        beq _fgresume   ;Do not try when column is 0, it freaks out
        lda $54
        beq _fgresume   ;if 0 then keys are done
        dec $54         ;dec # of keys left to place
        inc $51         ;actual num keys left
        lda ($d1),y     ;load byte
        ora #$c0        ;EBCM value for key!
        sta ($d1),y     ;store new value
        ;(end of injection)

_fgresume = *
_fgx    ldx $4a         ;Findgrow ypos
_fgy    ldy $49         ;Findgrow xpos
        inc $49         ;Next xpos (next round)
        cpy #$28        ; < 40
        beq _fgcr       ; next line if >= 40
        jsr cgloadcell  ; load cell byte
        beq _fgy        ; if 0 then get next xpos/byte
        sty $45         ;Set Current xpos
        stx $46         ;Set Current ypos
        jmp cangrowxy   ;Check if this can grow
_fgcr   lda #$00        ;Reset Findgrow xpos
        sta $49         ; 0->xpos
        inc $4a         ;Next Findgrow ypos
        lda $4a
        cmp #$19        ;check ypos (<25)
        bne _fgx        ;If we're at x40y25 we are ready to play!
        beq gameinit    ;Start game logic
```

That code also checks to see if we have completed generating the maze, at which point it enters the game logic. But before we jump into that I have to explain what happens when the "cangrowxy" routine discovers that it CAN GROW, instead of falling into the "findgrow" routine.

In the "cangrowxy" routine described a bit above there was the final line of:

```
_cgxy beq growloop ;if 0 then we can grow
```

This "growloop" routine is the place where the actual "growing" or "eating" occurs. This routine works by choosing a direction at random, and attempting to grow in that direction. It reuses the same "cgup" etc. routines that the "cangrowxy" uses because although we know we can grow in a direction, we don't know which direction. In a larger version of this algorithm I usually link all of this stuff together, but this is how it worked out in tinyrinth.

```
growloop = *
randdir = *
        ;jsr getrand; not a func, not reused yet
getrand = *
        lda #$80
        sta $028a ; Key Repeat; (injected here for #$80) just using #$80 for
                  ;smaller code
        sta $d412       ;sta $d404   d412 is V3, d404 is V1!!
        sta $d40f       ;set v3 random # gen bits
        lda $d41b       ; read random number from V3
        and #$03        ; Force Random number to 0-3
        clc
        adc #$01        ; Add 1 to get 1-4
        sta $4c         ; store rand direction
        ldy $45         ; Current Xpos
        ldx $46         ; Current Ypos
        jsr cangrow     ; Check if we can grow in that direction
        bne randdir     ; if <> 0 then Try again
        sta $4b         ; reset findgrow flag (injected here for .a==0)
grow = *
        ldx $4c         ;Get saved rand direction
        lda growinto,x  ; 1-4 (4, 8, 1, 2) Get bit set for new cell
        sta ($d1),y     ; write new location
        lda growfrom,x  ; 1-4 (1, 2, 4, 8) Get bit to set for old
```

```
        sta $4d         ; Save growfrom bit
        ldy $45         ; Reload Current xpos
        ldx $46         ; Reload Current ypos
        jsr cgloadcell  ; Load base cell again
        ora $4d         ; Combine with growfrom bit
        sta ($d1),y     ;Modify old cell to connect to new loc
        ;Change current position
        lda $4c         ; Get saved rand direction
        jsr cangrow     ; Get new x y again - (this will only perform next x/y
                        ;adj, returns <>0)
        sty $45         ; xpos   set to new location
        stx $46         ; ypos
        jmp popgridloop ; Return to populate grid loop
```

As you can see above I use the random number generator from V3 to get the random direction. A HUGE bonus here is I can use the value returned to directly call that main entry point in "cangrow" so it will figure out which direction to check by itself. If the direction it checks is bad it just loops and tries again. In a larger version of the program I'd make this much more efficient, but efficiency costs bytes. Many coders would be afraid of using a loop like this but since I pre-checked that I can grow before entering this loop, unless my random number generator never returns that direction, I should be fine!

Once we have a good direction I use the already set $d1 zp to read and update the screen data. Nice bonus of using the routine that sets it! The code that updates the data must update the cell you are growing to, but ALSO it must update the cell you are growing from, which is why I have to reload the original data up there. When it is finished doing the growing it moves the x/y positions ($45, $46) to the place it grew to; and resumes the maze generation from there.

As I said before, the findgrow routine will jump into the game logic once it knows it is finished, but before we talk about that I should describe another problem that I needed to solve.

In a 2 dimensional maze there are 16 possible pieces that can be formed. Starting with completely closed ending with completely open. I represent these pieces with numbers 0-15; each bit represents a side of a cell that is open/closed. So when I finish with the maze generation I have a grid of numbers 0-15. (Actually 1-14; in this game I never end up with 0's or 15's as all pieces are used.) In Tinyrinth the "grid" I used was the screen memory (40x25), so I ended up with a screen full of characters A-N.

Navigating a maze of characters A-N would be rather difficult so I had to decide whether to use the built in character set and transform what was on the screen or somehow use a character set. Well, another hitch was I wanted this to be a game and losing the simplicity of 0-15 for detecting which way a player could move would be bad - so I opted for a character set of 16 characters. Hmmm 8 bytes per character time 16 characters.. 128 bytes! Yikes!

Character Set Generation

So I was going to need a 128 byte character set, but I didn't want to have to lose 128 bytes. So what did I do? I coded about 70 bytes of code to generate the character set. This too pained me because I was sure I could make it smaller somehow, but once I got it working and a decent percentage smaller than the size of what it produced I stopped worrying about it.

Here is how it works: we know we want to represent the 16 pieces with 0-15 so the bits must give some hint to the shape of the piece. What I did was create a loop to check the current counter value and either set or skip the top, sides, bottom of the piece.

Here is how it works, I have commented each line of code to describe what it is doing.

```
        ; Generate Cset!
        lda #$20        ; write hi
        sta $48         ; use zp
        lda #$00        ; write lo

        ;Initialize Screen, variables (injected here to save bytes - using
        ;lda #$00)
        sta $51           ; Clear actual num keys placed counter (see findgrow)
        sta $d020
        sta EBCM0;  Set BG Color ($d021)
        ;(end of injection)

        tax             ; counter = 0
_again  sta $47         ; use zp
        ldy #$00        ; index
        txa             ; counter to a
        and #$01        ; check for top
        beq  ytop       ; yes top
```

```
        eor #%01111111 ; 00000001 -> 011111110 -> 10000001
_ytop   eor #%11111111 ; 00000000 -> 111111111
_6sides sta ($47),y    ; store top/sides to cset
        iny            ; next mem location
        txa            ; counter to a
        and #$02       ; check for right
        eor #%00000010 ; flip
        lsr            ; 00000010 -> 00000001 || 0->0
        sta $49        ; store for right side
        txa            ; counter to a
        and #$08       ; check for left side
        bne _noleft    ; no left
        eor #%10001000 ; 00000000 -> 10001000 -> 10000000
_noleft eor #%00001000 ; 00001000 -> 00000000
        ora $49        ; merge with right
        cpy #$07       ; 7->15->23->...
        bne _6sides    ; total of 6 side pieces
        txa            ; counter to a
        and #$04       ; check for bottom
        beq _ybot      ; no bottom
        eor #%01111010 ; 00000100 -> 01111110 -> 10000001
_ybot   eor #%11111111 ; 00000000 -> 11111111
        sta ($47),y    ; store bottom to cset
        inx            ; next counter
        clc            ; clear carry
        lda $47        ; inc zp
        adc #$08       ; by 8
        bne _again     ; do it again
        inc $48
        ldy $48
        cpy #$28       ;repeat through cset 2000-2800
        bne _again
        sty EBCM2      ;Set Death color ($d023) (using result of cset gen for
                       ;color value!)
```

I still think I could have made that smaller; either way though it was fun making a little routine to generate the cest. With that all said, we now know how the maze is generated. But now I needed a way to play it, as suggesting players use dry erase markers on their screens was not a great idea.

Game Loop

Ok, once "findgrow" reaches the end of its maximum size it branches to "gameinit". Game init sets up some quick things and gets going on the game play. The game loop loops until the player either gets all the keys or dies. When the player gets all the keys the level number ($53) is increased which is used to increase the maximum number of keys and the color of the maze!

Here is the game loop, I have commented much of the code and separated it into sections:

```
*** Flashes the keys, sets speed of "minitaur" based on the level

; Game Initialization and Game Loop
gameinit = *
gameloop=*
        inc EBCM3       ; Flash Keys ($d024)
        inc $58         ; Increase Speed counter #1 (0-255)
        bne moveplayer  ; Skip move
        inc $57         ; Increase Speed counter #2 ($57|#$f0 - 255)
        bne moveplayer  ; Skip Move

        ;set death speed
        lda $53         ;Use level for Speed value
        cmp #%11111000  ;If more than this use default speed
        bmi _dsp
        lda #%11111000  ;Default speed
_dsp    ora #%11110000  ;Set high nibble so counter counts up to 255
        sta $57         ; Set Speed counter #2

*** Moves the minitaur to the next position if it is time. (Checks for player
*** hits)

;move death
movedeath = *
        ldy $55         ;Baddy Xpos
        ldx $56         ;Baddy Ypos
        jsr cgloadcell  ; load the cell/point the zps (ANDs by #$1f)
        sta ($d1),y     ;store cleared value
_newy   iny             ;increase xpos
        cpy #$28        ;less than 40?
        bmi  go         ;don't reset
```

```
        ldx $46         ;ypos of player
        stx $56         ;ypos of death
        ldy #$00        ;clear xpos counter
_go     sty $55         ;Set baddy xpos
        lda #$ff        ;Get all bits! (see loadcell)
        jsr loadcell    ;load the cell/point the zps
        sta $59         ;Save cell value (withh all possible bits)
        and #%11000000  ;and by EBCM bits
        cmp #%11000000  ;Check for KEY - (so it can skip over)
        beq _newy       ;Jump ahead 1 more to skip key position
        cmp #%01000000  ;Check for player hit
        bne _nodie      ;Player is not dead
        jmp gameover    ;Game Over!
_nodie  lda $59         ;Reload stored value
        ora #$80        ;EBCM for Death
        sta ($d1),y     ;store value
; *** fall through to Move Player

*** Checks the keyboard for input, moves player if possible

;Move Player
moveplayer=*
_ffe4   jsr $ffe4       ;Get keypress
        beq gameloop    ;no key - goto gameloop
        and #%00000011  ;.a == 0-7 at this point
        tax
        lda keytodir,x  ;Loads from keytodir
;Move entity in game
; .a=direction 1-up 2-right 3-down 4-left
glmove  tax
        stx $4e         ; store direction
        lda growinto,x  ; get check bit
        sta $4f         ; store check bit
        ldy $45         ; current xpos
        ldx $46         ; current ypos
        jsr cgloadcell  ; load the cell (and with #$1f)
        sta ($d1),y     ; store the data (clear the EBCM)
        lda #$00        ; Bottom "fall out" fix
        sta $50         ; clear and of cangrow  Bottom "fall out" fix
        lda $4e         ; load direction
        jsr cangrow     ; call cangrow to move xpos/ypos
        and $4f         ; check bit
        bne glmyes        ; if we have a bit then we can move!
        ldy $45         ; reload xpos - do not move
        ldx $46         ; reload ypos - do not move
glmyes  lda #$ff        ; bits to obtain from loadcell
        jsr loadcell    ; load the cell/point the zps
        pha             ; temp store value for later checks
        and #$1f        ; clear other EBCM bits
        ora #$40        ; EBCM ORA Player/Baddy
        sta ($d1),y     ; store new data
        sty $45         ; store xpos of new position
        stx $46         ; store ypos of new position

*** Checks for key hits, minitaur hits, Check for end-level:
    (jumps to next level if it is the end of the level, loops to the beginning
     of the game loop if not.)

        ;Hit checks
        pla             ; load previous value
        and #$c0        ; check for hits "11xxxxxx"
        cmp #$c0        ; check for key hit
        bne _back       ;_notkey     ; to next check
        dec $51         ; dec number of keys left in level
        bne _back       ; if 0 then we should go to the next level
        jmp init        ; gen maze again
;_notkey cmp #$80        ; check for death hit!
;        bne _back
;        jmp gameover       ; game over
_back   jmp gameloop
        ;more checks here?

keytodir=*
.byte 2,1,4,3
```

    To make this all work, I reused the "cangrow" routine again! Because basically it is the exact same logic. One thing that I found unfortunate is I never coded a way for the minitaurs to have any intelligence, there was an attempt that worked - but it was FAR too many bytes (over by like 20-30). So I just axed it. I also at one time made the movement routines all one

subroutine that took parameters that would move any "entity". This also took too many bytes to be useful for this version of the game. Perhaps some day I'll write a kick butt 1k or 2k version with all those features coded in.

Various Space Saving Techniques Used

If you look throughout the code, you'll notice I comment a lot on "injections". These are little bits of code that are conveniently put in places to take advantage of the state of a register, or a memory location.
Here is an example: in the "getrand" routine I injected a line to set the key repeat, using the value that I wanted to also put into $d412 for the random number generation.

```
        lda #$80
        sta $028a; Ket Repeat; (injected here for #$80) just using #$80 for smaller code
        sta $d412       ;sta $d404   d412 is V3, d404 is V1!!
```

Another example, here I mixed some static data together to save some bytes and just labeled it so some would be reused. To save additional bytes I also positioned the label in front of the "rts". (This was because the code accessing the data never used an index of 0.) This is mixed with the rts because the first byte would be wasted

```
 growfrom = *
    rts
    .byte 1
; this is part growfrom part growto  48 is used for both
;  again first byte would be wasted so we over lap with the previous
growinto = *
    .byte 2, 4, 8, 1, 2

;explanation of above
; 0  1  2  4  8
;       0  4  8  1  2
;rts 1  2  4  8
;          4  8  1  2
```

I also reused the "cangrow" routine as much as possible, and on top of that I made it more usable by making multiple entry points so it could be used in different way.

Finally the best thing I did was use kernal routines, like $E544 to clear the screen to the set char color at $0286. This allowed me to change the color of the maze each level just by changing the value at $0286.
I also used $e9f0, Mapping the c64 says, "Set Pointer to Screen Address of Start of Line. This subroutine puts the address of the first byte of the screen line designated by the .X register into locations 209-210 ($D1-$D2)." This was handy because I could use this to read from and write to the screen.

Extended Color Background Mode

I could not use sprites. This made me sad, but there was just no room for it. So I just decided to allow the cset to repeat itself through the whole cset area ($2000-$2800) and use ECBM! This allowed me to just set a bit and make a maze piece a different color, what more when I read this value I could easily tell what the piece represented! That made collision detection very easy, and also allowed me to have cool flashing keys.

Conclusion

I had a LOT of fun coding this and look forward to another competition. I was surprised and amazed by the entries! This even inspired me to make a better version of Tinyrinth, of which I have not completed. I did add to it, I made a version in which the minitaurs move through the maze. I made another version that would also increase the number of minitaurs in the maze. I made yet another version that would change the size of the maze, and thought about changing the shape too. All of these ideas were easy to implement once I had the base game working.
Oh by the way, I know how to spell Minotaur, the Minitaur thing is just a bad joke. :)

Final Code Stats

Total source compiled: 445 lines in 1 file(s)
Symbol table: 29 global and 1 local constants, 0 global variables,
2 global and 15 local labels, 0 source labels.
Data size: 10 bytes
Code size: 475 bytes in 231 instructions
Memory block affected: $1000 - $11E4 (total size: 485 bytes)

Source

Note: This source is not the source I released for the event. It is a bit

smaller, I cannot locate my original source for some reason. Also, some of it
may differ a tiny bit from the code documented above. I had written the
article using the wrong copy of the source, I have attempted to go back and
update it though. (Note to self: keep better records!) :)

    Another note: This source compiles with c64asm by Balint Toth. :)

```
-- tinyrinth.asm  --
; Tinyrinth
;   entry for a 512b game contest
; Burning Horizon/FTA
; Mark Seelye
; mseelye@yahoo.com
; ===================================================================

* = $1000

;name   loc     desc       color  ecbm bits
EBCM0 = $d021 ; untouched  black   $00  00
EBCM1 = $d022 ; cursor     red     $40  01
EBCM2 = $d023 ; touched    green   $80  10
EBCM3 = $d024 ; keys       yellow  $c0  11

;ZPs used:  (Consolidation Possible if needed)
;43/44 - Not Used
;45/46 - Current X/Y Position, Maze Generation & Game
;47/48 - CSet Location, CSet Generation
;49    - Temp Storage, CSet Generation
;49/4a - Xpos/Ypos Findgrow, Maze Generation
;4b    - Flag, Findgrow, Maze Generation
;4c    - Temp Storage, randdir, Maze Generation
;4d    - Temp Storage, grow, Maze Generation
;4e/4f - Temp Storage, glmove, Game
;50    - Temp Storage, loadcell, Maze Generation & Game
;51    - NumKeys Left in level (affected by: destroyed & found keys)
;52    - not used
;53    - Current Level
;54    - # of Keys to try and place, gameinit, Game
;55/56 - X/Y Pos of Death
;57/58 - speed counter for death

;Collect all keys
; gen crummb path for visited areas?
; once all keys collected next maze is gen'd

; Initialization
setup = *
gameover = *
        ;Setup game variables
        ldx #$00
        stx $53         ; Start at level 1 (to be inc'd)
init = *
        ;Setup level
        inc $53         ; next level
        ldx $53
        stx $54         ;Counter for drawing Keys (next level)
        stx $0286       ;Character Color to use on clear (e544)

        ;Set Render Cursor Start Pos / Player Color
        lda #$05
        sta EBCM1       ;Set ebcm color PLAYER to GREEN ($d022)
        sta $45         ; Cursor/Player Position X (0-39)
        sta $46         ; Cursor/Player Position Y (0-24)
        ;Clear Screen
        jsr $E544       ;clear screen set to char color ($0286)

        lda #$5b
        sta $d011      ;turn on EBCM
        lda #$18
        sta $d018      ;Activate Cset

        ; Generate Cset!
        lda #$20       ; write hi
        sta $48        ; use zp
        lda #$00       ; write lo

        ;Initialize Screen, variables (injected here to save bytes - using lda #$00)
        sta $51          ; Clear actual num keys placed counter (see findgrow)
        sta $d020
        sta EBCM0;  Set BG Color ($d021)
        ;(end of injection)
```

```
        tax             ; counter = 0
_again  sta $47         ; use zp
        ldy #$00        ; index
        txa             ; counter to a
        and #$01        ; check for top
        beq _ytop       ; yes top
        eor #%01111111 ; 00000001 -> 011111110 -> 10000001
_ytop   eor #%11111111 ; 00000000 -> 111111111
_6sides sta ($47),y     ; store top/sides to cset
        iny             ; next mem location
        txa             ; counter to a
        and #$02        ; check for right
        eor #%00000010 ; flip
        lsr             ; 00000010 -> 00000001 || 0->0
        sta $49         ; store for right side
        txa             ; counter to a
        and #$08        ; check for left side
        bne _noleft     ; no left
        eor #%10001000 ; 00000000 -> 10001000 -> 10000000
_noleft eor #%00001000 ; 00001000 -> 00000000
        ora $49         ; merge with right
        cpy #$07        ; 7->15->23->...
        bne _6sides     ; total of 6 side pieces
        txa             ; counter to a
        and #$04        ; check for bottom
        beq _ybot       ; no bottom
        eor #%01111010 ; 00000100 -> 01111110 -> 10000001
_ybot   eor #%11111111 ; 00000000 -> 11111111
        sta ($47),y     ; store bottom to cset
        inx             ; next counter
        clc             ; clear carry
        lda $47         ; inc zp
        adc #$08        ; by 8
        bne _again      ; do it again
        inc $48
        ldy $48
        cpy #$28        ;repeat through cset 2000-2800
        bne _again

        sty EBCM2       ;Set Death color ($d023) (using result of cset gen for color value!)


popgridloop = *
        ;can grow from current?
        ldy $45          ;xpos
        ldx $46          ;ypos
        ;Can grow in any direction?
cangrowxy = *
        jsr cgup         ;check up
        beq _cgxy        ;if 0 then we can grow
        inx              ;offset up check
        jsr cgright      ;check right
        beq _cgxy        ;if 0 then we can grow
        dey              ;offset right check
        jsr cgdown       ;check down
        beq _cgxy        ;if 0 then we can grow
        dex              ;offset down check
        jsr cgleft       ;check left
_cgxy   beq growloop     ;if 0 then we can grow
; *** fall into findgrow
findgrow = *
        lda $4b          ; Check byte 0 != resume findgrow
        bne _fgresume
        sta $49          ;Reset Findgrow Xpos
        sta $4a          ;Reset Findgrow Ypos
        inc $4b          ;Set findgrow flag to resume (<>0)

        ;Place keys in corners (injected here for ease of placement, d1/d2 is pointed at a dead
end)
        iny              ; offset left check
        beq _fgresume    ;Do not try when column is 0, it freaks out
        lda $54
        beq _fgresume    ;if 0 then keys are done
        dec $54          ;dec # of keys left to place
        inc $51          ;actual num keys left
        lda ($d1),y      ;load byte
        ora #$c0         ;EBCM value for key!
        sta ($d1),y      ;store new value
        ;(end of injection)
```

```
_fgresume = *
_fgx    ldx $4a         ;Findgrow ypos
_fgy    ldy $49         ;Findgrow xpos
        inc $49         ;Next xpos (next round)
        cpy #$28        ; < 40
        beq _fgcr       ; next line if >= 40
        jsr cgloadcell  ; load cell byte
        beq _fgy        ; if 0 then get next xpos/byte
        sty $45         ;Set Current xpos
        stx $46         ;Set Current ypos
        jmp cangrowxy   ;Check if this can grow
_fgcr   lda #$00        ;Reset Findgrow xpos
        sta $49         ; 0->xpos
        inc $4a         ;Next Findgrow ypos
        lda $4a
        cmp #$19        ;check ypos (<25)
        bne _fgx        ;If we're at x40y25 we are ready to play!
        beq gameinit    ;Start game logic


growloop = *
randdir = *
        ;jsr getrand; not a func, not reused yet
getrand = *
        lda #$80
       sta $028a; Ket Repeat; (injected here for #$80) just using #$80 for smaller code
        sta $d412       ;sta $d404   d412 is V3, d404 is V1!!
        sta $d40f       ;set v3 random # gen bits
        lda $d41b       ; read random number from V3
        and #$03        ; Force Random number to 0-3
        clc
        adc #$01        ; Add 1 to get 1-4
        sta $4c         ; store rand direction
        ldy $45         ; Current Xpos
        ldx $46         ; Current Ypos
        jsr cangrow     ; Check if we can grow in that direction
        bne randdir     ; if <> 0 then Try again
       sta $4b          ; reset findgrow flag (injected here for .a==0)
grow = *
        ldx $4c         ;Get saved rand direction
        lda growinto,x  ; 1-4 (4, 8, 1, 2) Get bit set for new cell
        sta ($d1),y     ; write new location
        lda growfrom,x  ; 1-4 (1, 2, 4, 8) Get bit to set for old
        sta $4d         ; Save growfrom bit
        ldy $45         ; Reload Current xpos
        ldx $46         ; Reload Current ypos
        jsr cgloadcell  ; Load base cell again
        ora $4d         ; Combine with growfrom bit
        sta ($d1),y     ;Modify old cell to connect to new loc
        ;Change current position
        lda $4c         ; Get saved rand direction
        jsr cangrow     ; Get new x y again - (this will only perform next x/y adj, returns <>0)
        sty $45         ; xpos   set to new location
        stx $46         ; ypos
        jmp popgridloop ; Return to populate grid loop


; Game Initialization and Game Loop
gameinit = *
gameloop=*
        inc EBCM3       ; Flash Keys ($d024)
        inc $58         ; Increase Speed counter #1 (0-255)
        bne moveplayer  ; Skip move
        inc $57         ; Increase Speed counter #2 ($57|#$f0 - 255)
        bne moveplayer  ; Skip Move

        ;set death speed
        lda $53         ;Use level for Speed value
        cmp #%11111000  ;If more than this use default speed
        bmi _dsp
        lda #%11111000  ;Default speed
_dsp    ora #%11110000  ;Set high nibble so counter counts up to 255
        sta $57         ; Set Speed counter #2


;move death
movedeath = *
        ldy $55         ;Baddy Xpos
        ldx $56         ;Baddy Ypos
        jsr cgloadcell  ; load the cell/point the zps (ANDs by #$1f)
        sta ($d1),y     ;store cleared value
```

```
_newy   iny             ;increase xpos
        cpy #$28        ;less than 40?
        bmi _go         ;don't reset
        ldx $46         ;ypos of player
        stx $56         ;ypos of death
        ldy #$00        ;clear xpos counter
_go     sty $55         ;Set baddy xpos
        lda #$ff        ;Get all bits! (see loadcell)
        jsr loadcell    ;load the cell/point the zps
        sta $59         ;Save cell value (withh all possible bits)
        and #%11000000  ;and by EBCM bits
        cmp #%11000000  ;Check for KEY - (so it can skip over)
        beq _newy       ;Jump ahead 1 more to skip key position
        cmp #%01000000  ;Check for player hit
        bne _nodie      ;Player is not dead
        jmp gameover    ;Game Over!
_nodie  lda $59         ;Reload stored value
        ora #$80        ;EBCM for Death
        sta ($d1),y     ;store value
; *** fall through to Move Player

;Move Player
moveplayer=*
_ffe4   jsr $ffe4       ;Get keypress
        beq gameloop    ;no key - goto gameloop
        and #%00000011  ;.a == 0-7 at this point
        tax
        lda keytodir,x  ;Loads from keytodir
;Move entity in game
; .a=direction 1-up 2-right 3-down 4-left
glmove  tax
        stx $4e         ; store direction
        lda growinto,x  ; get check bit
        sta $4f         ; store check bit
        ldy $45         ; current xpos
        ldx $46         ; current ypos
        jsr cgloadcell  ; load the cell (and with #$1f)
        sta ($d1),y     ; store the data (clear the EBCM)
        lda #$00        ; Bottom "fall out" fix
        sta $50         ; clear and of cangrow  Bottom "fall out" fix
        lda $4e         ; load direction
        jsr cangrow     ; call cangrow to move xpos/ypos
        and $4f         ; check bit
        bne glmyes        ; if we have a bit then we can move!
        ldy $45         ; reload xpos - do not move
        ldx $46         ; reload ypos - do not move
glmyes  lda #$ff        ; bits to obtain from loadcell
        jsr loadcell    ; load the cell/point the zps
        pha             ; temp store value for later checks
        and #$1f        ; clear other EBCM bits
        ora #$40        ; EBCM ORA Player/Baddy
        sta ($d1),y     ; store new data
        sty $45         ; store xpos of new position
        stx $46         ; store ypos of new position

        ;Hit checks
        pla             ; load previous value
        and #$c0        ; check for hits "11xxxxxx"
        cmp #$c0        ; check for key hit
        bne _back       ;_notkey     ; to next check
        dec $51         ; dec number of keys left in level
        bne _back       ; if 0 then we should go to the next level
        jmp init        ; gen maze again
;_notkey cmp #$80        ; check for death hit!
;        bne _back
;        jmp gameover       ; game over
_back   jmp gameloop
        ;more checks here?

keytodir=*
.byte 2,1,4,3


;Check if a cell can grow a direction
;1-up 2-right 3-down 4-left
; (y xpos, x ypos, a=dir)  x/y switched for indirect lda($xx),y below
; return: a == 0 : true (can move)
;         a <> 0 : false (can not move)
cangrow = *
    cmp #$01
    beq cgup
```

```
    cmp #$02
    beq cgright
    cmp #$03
    beq cgdown
    ;cmp #$04
    ;beq cgleft *** not needed falling through
cgleft = *
    dey         ;set xpos - 1
    cpy #$ff    ;check xpos
    beq cgno
    bne cgloadcell
cgright = *
    iny         ;set xpos + 1
    cpy #$28    ;check xpos (<40)
    beq cgno
    bne cgloadcell
cgup = *
    dex         ;set ypos - 1
    cpx #$ff    ;check xpos
    beq cgno
    bne cgloadcell
cgdown = *
    inx         ;set ypos + 1
    cpx #$19    ;check ypos (<25)
    beq cgno
    ;*** fallthrough, bne cgloadcell not needed
cgloadcell = *
    lda #$1f
loadcell = *        ;x = ypos, y = xpos, a = and value
    sta $50
    jsr $e9f0   ; sets $d1 $d2
;59888          $E9F0
;Set Pointer to Screen Address of Start of Line
;This subroutine puts the address of the first byte of the screen line
;designated by the .X register into locations 209-210 ($D1-$D2).
    lda ($d1),y     ;load byte  (x pos in y reg!)
cgno = *
    and $50  ;#$1f = use only low 5 bits!
    ;rts see below!
;  This is mixed with the rts because the first byte would
;  be wasted
growfrom = *
    rts
    .byte 1
; this is part growfrom part growto  48 is used for both
;  again first byte would be wasted so we over lap with the previous
growinto = *
    .byte 2, 4, 8, 1, 2

;explanation of above
; 0  1  2  4  8
;       0  4  8  1  2
;rts 1  2  4  8
;          4  8  1  2



;Notes

; 1
;8 2
; 4
;
;
;*** * * *** * *
;*0* *1* *2  *3   @  A  B  C
;*** *** *** ***
;
;*** * * *** * *
;*4* *5* *6  *7   D  E  F  G
;* * * * * * * *
;
;*** * * *** * *
; 8*  9*  a   b   H  I  J  K
;*** *** *** ***
;
;*** * * *** * *
; c*  d*  e   f   L  M  N  O
;* * * * * * * *
;

-- end: tinyrinth.asm  --
```

Tetrattack!

by Stephen L. Judd <sjudd@ffd2.com>

Introduction

The main point of Tetrattack!, of course, was to try and write a 3D engine in 512 bytes. I had a few other routine ideas but couldn't think of anything remotely fun to do with them, but I finally thought of a neat 3D game idea and decided to go for it. Alas, as code kept getting bigger that idea had to be discarded, and the eventual result was tetrattack, the simplest/smallest 3D game that wasn't (totally) stupid.

But it is a 3D engine in 512 bytes!

At first blush a 512-byte 3D engine might seem ridiculous but think about it: a basic 3d program doesn't need to store any graphics definitions or things like that -- it draws its own as it goes. In fact, all that is needed is a line routine, a rotation/projection routine, and some routines to keep track of angles and positions and such. What could possibly be hard about that, right?

Well, it seemed reasonable at the time, anyways. At exactly 512 bytes, the code really had to sweat out the bytes, and every single byte saved was important; meanwhile, the routines still had to be reasonably fast. So this article will discuss the line routine, the rotation/projection routine, and the object and game manipulation routines, and the various optimizations used.

I must say that writing this program really was fun; I'm sure that everyone else had the same problems I had, trying to shift from a "cycle-optimization" mindset to a "byte-optimization" mindset. Finding those extra few bytes here and there was a mighty challenge, sometimes taking hours or even days just to save one or two bytes. That part was frustrating, of course, as is the fact that the 'game' aspect of the program is thoroughly lame. But it was a neat challenge, and there was never an issue of adding too many features to the program and never finishing it!

Quick Summary

A full-on 3D program uses matrix multiplications to compute rotations, but 3D calculations get a whole lot easier in a Battlezone or driving-type situation, when you only turn left or right. There are no longer rotation matrices to keep track of -- just a single angle. This means that rotations are done with a simple INC/DEC, and there are no matrix multiplies to deal with. Rotations are done with some simple table lookups.

The line drawing routine is pretty standard. Self-modifying code is used to manipulate a single routine to draw different lines, by swapping INX/INY instructions (which turns out to be less efficient, memory-wise -- sigh, oh well). Lines are plotted to a double-buffered 128x128 charmap, and once again setting up the screen took extra bytes.

The "game code" is the simplest possible -- tetrahedrons are moved by just incrementing their z-coordinate, and rotated by incrementing their angle. Their locations are determined by whatever zero page is initialized to -- the program never sets explicit locations. Instead, I just chose a portion of zero page that had a reasonable range of default values (default zero-page values are used extensively in the code)

In addition to default zero-page values the code uses several kernal routines (including a few unusual ones, for example the shift-C= routine for swapping charset buffers), self-modifying code, lots of zero-page instructions, and a whole bunch of tightly optimized code.

The code was written over a few weeks -- a good week of planning and writing out possible routines, and a good week or two of coding.

A couple of times I froze the program (using AR) and the freeze almost always happened in the same place: the buffer clearing routines. So the rotation and line drawing routines turned out to be fast enough, even with all the

byte savings.

The code was developed using the Sirius assembler on a 128D+SCPU -- and btw, the Jammon debugging mode has turned out to be awfully cool, with being able to visually single-step through code while simultaneously viewing memory and registers. Action Replay was downright clunky afterwards (but could freeze!).

Yep, just a plug, with a bit of amazement that it actually all worked.

I also tested it with VICE, to make sure it worked OK. VICE initializes zero page to different values than my 128, unfortunately, so some tweaking had to be done.

And that's about it. Now on to more detail.

Setting up

The setup routines initialize VIC, initialize tables used by the line and rotation routines, and set up the screen. I note that when I started planning this program I computed the sizes of line and rotation routines without considering the code needed to set up tables and such for those routines. Oops!

The default initial values for .X, .Y, and .A are 0 after a SYS, so the code made a little use of this.

Line routine setup

The line routine uses tables to look up pixel addresses and bit locations for each x,y coordinate (the line routine stores the x and y coordinates directly in the .X and .Y registers, so looking up pixel addresses amounts to LDA Bitp,X kinds of instructions). The "bitmap" is a 128x128 charmap, so the calculations are pretty easy. The bit patterns which correspond to different x-coordinates (10000000 01000000 etc.) are computed using

```
        cmp #$80
        rol
```

to cyclically rotate a single bit.

Rotation routine setup

Another little piece of code is piggybacked into this loop, which extends a sine table from 0..pi. In this program, "angles" can go from 0..63, corresponding to 0..2*pi; the most memory-efficient way to store the sine table is to store the values for 0..pi/2, and then extend the table to pi/2..pi. That is, angle=16 (angles go from 0..63, remember) corresponds to pi/2, so the job of extending the table amounts to copying table values 15..0 to values 17..32 (value 16 is not copied because it is pi/2 -- the values are 'mirrored' through pi/2. To put it another way: where would it be copied to?).

This amounts to starting one index register at 15 and decrementing it, while starting the other register at 17 and incrementing:

```
        ldy #15
        ldx #17
        lda table,y
        sta table,x
        inx
        dey
```

The problem here is piggybacking into a loop where .Y is initialized to ldy #$7F instead of ldy #15. The solution is to initialize .X carefully

```
        [ldy #$7f]
        ldx #$90
```

and use the following:

```
        lda sin0,y
        sta sin0+17,x
        inx
        [dey]
```

The idea is that when .Y=15, .X will equal 0 and start incrementing. This

copies a lot of junk into the table that is never used but doesn't overwrite the existing 0..16 table values, and saves several bytes.

The rotation tables are tables of r*sin(theta), one table for every allowed value of theta=0..64 and r=-128..127. The idea is that every 256-byte page contains r*sin(theta) for one value of theta, so that a table lookup amounts to

```
        lda theta
        ora #>high byte of sin tables
        sta zp+1
        ldy r
        lda (zp),y
```

The table values are computed using a standard shift-and-add multiply routine, with a little extra code to handle negative values of r. What about negative values of sin(theta), though?

Recall that theta=0..63 corresponds to 0..2*pi, but the init code only extends the sine table from 0..pi (i.e. theta=0..32). The table values for pi..2*pi are computed simply by taking the negative of the values for 0..pi:

```
         sta (p4000),y
         eor #$ff
         clc
         adc #1
         STA (p6000),y
```

(The clc adc #1 code was painful, byte-wise, but necessary). The pointers p4000 and p6000 are initialized to $4000 and $6000 by default -- or at least are supposed to be. Unfortunately, VICE and my 128D had different ideas about this, so a little extra code was necessary to initialize them.

A big reason for putting a table at $6000 is that it ends at $8000, to provide an easy exit condition for the loop:

```
         inc p6000+1
         bpl :mult8       ;to $8000
```

Screen setup

A couple of things need to be done here: clear the screen, set the color RAM to the background color, and put a 16x16 block of characters in the middle of the screen (and do so in as few bytes as possible!).

Clearing the screen is a piece of cake:

```
         sta $d021
         sta $0286        ;clear color
         jsr $e544        ;clr scr
```

The $0286 STA is to clear the color RAM to black, and work with different kernals. (With both color RAM and background the same, pixels won't appear where they aren't wanted).

Storing a block of chars isn't quite so straightforward; rather, the straightforward methods use up a fair number of bytes, especially since both screen and color RAM have to be set up. Here's the method I finally came up with:

```
:l2a
         jsr $e8ea        ;scroll up
                          ;.A = ($AC)
                          ;.X=00
         clc              ;necessary?
:l2                       ;set one line
         sta $0720+12,x
         inc $db20+12,x
         inx
         adc #16
         bcc :l2
                          ;         adc #$00
         inc $ac

         cmp #15
         bne :l2a
```

Kernal routine $e8ea is the routine to scroll the entire screen up one line, and disassembly shows that when it exits .A is equal to the contents of $AC

and .X is 0. So the routine stores one line of chars to the middle of the screen, increments the corresponding color RAM (to white), and moves the screen up (chars and color RAM). By incrementing $AC before moving the screen up ($AC starts at zero normally), .A is re-initialized to its previous value+1, so this prints the next line of chars to the screen (that is, the previous line of chars + 1). By doing this 16 times a 16x16 block of chars is produced, with color RAM set to white inside the block, and set to black everywhere else.

And that brings us to the main program loop.

Main Loop

The main loop is pretty straightforward: clear buffer, get input, update positions and angles, render the buffer, and swap buffers.

The only thing special in this process is swapping buffers via the kernal routine at $eb59 -- this is the routine that is called when C= and shift are held down.

There is also a lame dot that pulses to the screen when you hold the fire button down (how cool some laser lines would have been). Alas...

But that's pretty much it. So now let's talk a little about some of the optimizations that can be made for a 3D program, followed by a discussion of the line routine and the rotation routine.

3D Stuff

For detailed information on how to "do" 3D worlds I suggest reading issue #16 of C=Hacking, but the basics of rendering a 3D world are:

1) figuring out where an object is, relative to you,
2) if the object is visible, computing what it looks like from your
   perspective, and
3) rendering the object.

To define and operate on an "object", we need to know the vertices of the object, and how to connect them together with lines, i.e. to draw the edges. The simplest object of all is a tetrahedron -- four points, and every point is connected to every other point. This turns out to be pretty important, as a lot of memory is saved by using only tetrahedrons. To see why, first consider what the code to draw edges from a list of vertices must look like:

```
        rotate points

        ldx #number_of_edges
:loop   stx temp
        ldy edge_vertex1,x      ;index into some list of vertices
        lda edge_vertex2,x
        tax
        jsr drawline
        ldx temp
        dex
        bpl :loop
```

Now consider a cube; in my original code idea, I was going to use cubes and tetrahedrons (pyramids). A cube has eight vertices and 12 edges. Each vertex has an x, y, and z coordinate, and from the code above each edge needs two bytes, one for each vertex of the edge. 48 bytes, just to store a cube -- almost 1/10 of the entire 512-byte code! (Using code to generate the vertices is possible, but still takes a fair number of bytes.)

Compare this to the humble tetrahedron, which only requires twelve bytes: three for each vertex. What about the edges? Remember that every point is connected to every other point in a tetrahedron. This means that we do not have to store the edge connections, but can compute them manually, and draw the whole thing using the following code:

```
         [count = 3]
                        ; Plot
:ploop   ldy count
:l2      ldx count
         dey
         tya
         pha
```

```
         jsr DrawLine
         pla
         tay
         bne :l2
         dec count
         bne :ploop
```

This code simply draws lines between all vertices: 3-2, 3-1, 3-0, and so on. It uses pretty much the same number of bytes as the earlier code, but doesn't require any storage of the edge connections (12 bytes to store the connections).

Next consider the vertices of a tetrahedron. It turns out that by carefully selecting the vertices, default zero-page values can be used, something that isn't possible with e.g. a cube. Consider the following tetrahedron vertices:

```
Px = 0, -4, 16, -4
Py = 16, 0,  8,  0
Pz = 0, 16,  0, -16
```

(just the vertices of a more or less regular tetrahedron which I drew on a piece of paper). The trick is to arrange these coordinates in the right way.

Location $ae/$af is used by the load routine (among others) to store to memory while loading; at the end of a load, this location contains the end address+1. And, it is surrounded by zeros:

```
>C:00ab  00 00 00 00  00 00 00 3c  03 00 00 00  00 00 00 00   .......<........
```

which means that by engineering the end address right we can store two coordinates here. In this case, I chose the end address+1 to be $10f0, which stores $f0 and $10 in $ae/$af -- -16 and 16, two of the Pz values, the other two being zero.

This was my original plan anyways, until I re-discovered that the screen setup routine altered the value of $ac (the kernal routine to move the screen up exits with .A=$ac, so incrementing it re-initializes .A in every loop iteration.) On the other hand, it increments it 16 times, so that after the load and the init the zp values are

```
>C:00ab 00 10 00 f0  10 ..
```

The values 00 10 00 f0 are exactly the Pz values above: 0, 16, 0, -16. Careful study of default zero-page values then shows

```
>C:0040  00 00 08 00  00 00 00 24  00 00 00 00  00 00 00 00   .......$........
```

This is pretty much Py, except for the first value -- so all the code has to do is store a 16 at location $40, and it turns out the screen init loop ends with .X=$10. So, one STX does the job, saving yet more bytes. Moreover, by locating the vertices in zero page, zero-page instructions can be used to access the point lists, saving even more bytes.

Now, the above are the vertices, but what about the _centers_, i.e. the object locations? Once again, to save space I just used some default zero-page values, with the x-coordinates at $14 and the z-coordinates at $7d. The reason these were chosen is pretty straightforwards. Since objects just move 'downstream' with increasing z-coordinate, the z-coords should be somewhat spread out; the values at $7d are:

```
>C:007d  3a b0 0a c9  20 f0 ef 38  e9 30 38 e9  d0 60 80 4f
```

which are somewhat spread-out. The x-coords, on the other hand, need to be clustered around 0. The camera (you) is located at x=0, and if objects have a large value for the center x-coordinate they are way off to the side (you've probably noticed a few objects like this). Location $14 has some zeros and some nonzero numbers that are near zero, i.e. that cluster around zero:

```
>C:0014  00 00 19 16  00 0a 76 a3  00 00 00 00  00 00 76 a3
```

Not great values, but good enough -- beggars can't be choosers.

Of course, poking different values into locations 20 and above will put the tetrahedrons in different places.

Rotation and Projection

Rotations are done very simply, using tables. As outlined earlier, there is a complete set of tables of r*sin(theta). At first blush, the 'natural' way of setting up these tables would be to have a table of 1*sin(theta) in one page, a table of 2*sin(theta) in the next page, and so on, maybe fitting a table of cos(theta) in the same page. The location of r*cos(theta) would then be given by

```
        page = offset+256*r (e.g. $40+r for tables starting at $4000)
        value = page+theta
```

It makes a lot more sense, though, to instead let _theta_ be the page index, and let each page contain r*sin(theta) for different values of _r_; the lookup is then

```
        page = offset+256*theta (e.g. $5300 for theta=$13)
        value = page+r
```

For one thing, the pointer setup is very easy -- there's only one value of theta to set up, instead of different values of r (for different x,y coordinates). For another, the entire range r=-128..127 can be used, which means the same tables can be used to rotate object _centers_ in addition to object _vertices_.

(If you recall programs like lib3d, the object centers are 16-bit signed values and have their own rotation routine, whereas the object vertices are limited to -96..95 and have a separate, optimized rotation routine; in this case, both the vertices and centers are eight-bit signed numbers.)

This all means that rotations (e.g. y*cos(theta) - x*sin(theta)) amount to

```
         LDY Py,X
         LDA (cos),Y
         LDY Px,X
         SEC
         SBC (sin),Y
```

to rotate a point.

Now, the usual 3D world calculations look like:

```
        rotate object centers
        rotate vertices, add to rotated center, and project
```

Consider rotating the z-coordinate; the code will look like

```
         LDA (cos),Y
         LDY Px,X
         SEC
         SBC (sin),Y
         CLC
RotM1    ADC CZ
```

that is: rotate vertex (z*cos(theta) - x*sin(theta)) and add to center (ADC CZ). Here's the great part: if we instead want to rotate the _center_ of the object, we want to _store_ the rotated point in CZ; the code would look like

```
         LDA (cos),Y
         LDY Px,X
         SEC
         SBC (sin),Y
         STA CZ
```

This is exactly the same code as above, except the ADC is changed to a STA. Therefore only one routine is needed, using self-modifying code to decide whether to STA the center coordinate or ADC it. In the code, the appropriate instruction is passed into the routine in the .Y register.

There's one other thing to be mentioned about the rotation routine. The rotation tables are of r*sin(theta), and r*cos(theta) is of course done by adding pi/2 to the theta-coordinate (theta+16 in the program). There are two ways to do this -- add 16 to the theta coordinate and correct any overflow (AND #63, for example), or extend the tables by an extra pi/2.

Byte-wise, the first method is the best, but check this out: instead of adding pi/2, we can also subtract 3*pi/2. The setup code then becomes:

```
RotProj
         AND #$3F
```

```
         ORA #>SINTAB
         STA sin+1
         SBC #$2F          ;-3*pi/4
                           ;         ADC #$10
                           ;         AND #$3F
         ORA #>SINTAB
         STA cos+1
```

The trick works here because of the ORA #>SINTAB, which is $4000 -- this allows negative angles to work, by borrowing higher bits. Consider:

.A	ORA #>SINTAB	SBC #$2F	ORA #>SINTAB
$12	$52 - sin tab at $5200	$22	$62 - cos tab at $6200
$22	$62 - sin tab at $6200	$32	$72 - cos tab at $7200
$32	$72 - sin tab at $7200	$42	$42 - cos tab at $4200

(C is clear in the SBC above). By using an SBC, angles automatically wrap around from $8000 to $4000. This saves two bytes over the ADC #$10 AND #$3F method (commented out above). Every byte counts!!!

Projections

The easiest way to do projections, once again, is via tables of f(x,z)=x/z, much like the tables of f(r,theta) = r*sin(theta). Alas, the setup code to generate these tables took up too much space, so the projects are computed 'manually', using a shift-and-add division routine.

Now, it turns out that the routine really computes 64*x/z -- the 64 is the magnification factor. The nice thing about a shift-and-add routine is that multiplying by 64 is possible just by changing the number of shifts. So the routine adds an extra 6 shifts into the division loop -- a little inefficient cycle-wise, but pretty thrifty byte-wise.

Drawing lines

Obviously, to render an object we need a line drawing routine. The line routine in the program is pretty standard, and described in several earlier issues of C=Hacking, so I won't go into too much detail. Most of the paragraphs below are just random tidbits, so feel free to skip any (there won't be a test).

The simplest and most efficient routines draw to a charmap -- you can read about this in detail in C=Hacking #8 and beyond -- since the .Y register maps directly to the y-coordinate, and the column-offsets are easy to compute. The biggest charmap available is 128x128, so that's what I used.

Overall, the line routine is pretty small, while being reasonably fast. The plot routine is 'dumb', and recalculated for every pixel, but the recalculation itself is fairly efficient, with a few table lookups. It draws into a 128x128 buffer and can handle off-screen coordinates, and the buffer can be anywhere (that it, it supports a double-buffer display). So, I think it turned out OK, all things considered.

With that background, here's the main line drawing loop:

```
plot
         pha
         tya
         bmi calcline
         lda bitphi,x
         ora base
         sta point+1
         lda bitplo,x
         sta point
         lda bitp,x
         beq calcline
         ora (point),y
         sta (point),y
calcline
         pla
mod2     sbc #dx
         bcs mod5
mod3     adc #dy
mod4     inx
mod5     iny
         dec temp
```

```
         bne plot
```

The basic line routine iteration goes like

```
        plot x,y
        y=y+1
        a = a - dx
        if a<0 then a=a+dy, x=x+1
```

for slope>1. The code for slope<1 is exactly the same, except for swapping x and y, and dx and dy. Therefore, I just used one routine and self-modifying code to set INX and INY etc. in the right places (mod2, mod3, mod4, and mod5 above). In retrospect this was not smart, and I think it would have saved 5 or 6 bytes to use two routines, and jsr to the plot routine -- the routines to set up the self-modifying lines take more space than the basic iteration.

The x and y coordinates are stored directly in the .x and .y registers. This makes the plotting really easy and relatively fast, and works well with the 128x128 charmap.

Off-screen points are a must, so the allowed coordinates are -64..192, giving 64 pixels to either side of the 128x128 visible area. Checking for these is easy, since the high bit will be set for coords <0 and >127. These are checked in the plot routine. The y-coordinate is checked directly, using tya; the x-coordinate is checked for range by use of the bitp table -- this table contains the bit pattern for the x-coordinate (%1000000 %01000000 etc.), and all of the values for x>127 are set to zero. Once again, in retrospect it would have been more memory-efficient to do this with txa (two more bytes), but darn it all, SOMETHING has to be at least a little bit cycle efficient here! Old habits die hard!

The line routine always draws from left to right, which means the initialization routine calculates dx = x2-x1, and if dx<0 then it swaps the two endpoints. This creates a problem when coordinates are -64..192 though; after subtracting x2 and x1 do you check the negative bit, or the carry bit? Cases like x1=-1, x2=1 should work, so the carry bit is useless; on the other hand, long lines, say x1=1, x2=130, should also work, so the high bit is also useless. The solution I used was to offset coordinates by +64 (and once again, in retrospect...).

The point coordinates are stored in two lists, one of x-coords and one of y-coords. This means that the endpoints are simply indexes into those point lists, i.e.:

```
        xlist contains list of x-coords; ylist list of y-coords
        xlist,x and ylist,x contain left endpoint
        xlist,y and ylist,y contain right endpoint
```

That simplified a lot of things -- the lists can be in zero-page, swapping coordinates is very simple, and so on.

The routine is double-buffered -- flickery single-buffers are just too painful. Since $2000 and $2800 are the locations of the two charset buffers, the current buffer is stored in zero-page variable $ce ("base" in the plot code above), which starts out initialized to #$20 (location $81 would also work) -- one more variable to not initialize.

So, in summary: a pretty standard line routine, with a few memory optimizations thrown in.

Wrap-up

Well, that's about it. As you can see, the code really runs on the edge -- zero-page has to be initialized correctly, little bits and pieces are missing from certain routines, things have to line up just right... but it works! There are undoubtably other places where bytes can be saved, of course, but I think it does a pretty good job of shaving bytes overall.

See you at next year's contest!

```
*
* tetrattack!
*
* Mini-3D rendering package
*
```

```
* - Motion in the x-z plane => rotations about one axis only
* - Plots to charmap
*
* SLJ 9/01
*
                              ;         org $1000
         org $0ef0

NUMOBJS  = 9

temp     = $02
dx       = $03
dy       = $04
point    = $05

AUX      = $03
ACC      = $04

CX       = $07
CZ       = $08
rottemp  = $09            ;0 initially

zpoint   = $0a            ;low byte already set to zero
zpoint2  = $0c

count    = $0e
angle    = $0f

cenx     = $14

Py       = $40

XCoord   = $50
YCoord   = $60

theta    = $70

cenz     = $7d

                          ; $ac taken by screen setup routine
                          ; final value = $10

Pz       = $ab


base     = $ce            ;$20 initially
                          ;alternate: ($81)

p4000    = $8f            ;$4000 initially
p6000    = $89            ;$60xx initially
p8000    = $9c            ;$8000

tagged   = $e0

sin      = $f7            ;0 low byte
cos      = $f9

curobj   = $ff


bitp     = $1c00
bitplo   = $1d00
bitphi   = $1e00
bitphi2  = $1f00
SINTAB   = $4000
PROJTAB  = $8000


DEY      = $88
INY      = $c8
INX      = $e8
ADC      = $65
STA      = $85

start
         sta $8f          ;set up p4000

*
* Set up VIC
*
```

```
         sta $d021
         sta $0286        ;clear color
         jsr $e544        ;clr scr
         lda #$18
         sta $d018


*
* Line drawing tables
*

         ldx #$90
         ldy #$7f
         lda #$01
:l1      sta bitp,y
         cmp #$80
         rol

         pha
         tya
         lsr
         lsr
         lsr
         lsr
         sta bitphi,y
         lda #00
         sta bitp+128,y   ;0 = dont draw
         ror
         sta bitplo,y

*
* Extend sin table to 0..31
*
         lda sin0,y
         sta sin0+17,x
                          ;sta sin0+32,x
         inx

         pla

         dey
         bpl :l1
                          ;.y=$ff
                          ;.x=$10

*
* Set up sin tables
*
InitRot

         iny
         sty p6000        ;rats :(

                          ;         ldy #00      ;rats :(
                          ;
                          ;         sty temp
                          ;:l3      ldx rottemp
                          ;         lda sin0,x
                          ;         sta aux      ;sin(theta)
                          ;
                          ;:l4
                          ;
                          ;         clc
                          ;         jsr MULT8    ;.Y * AUX

                          ;
                          ; AUX*.Y -> ACC,.A (low,hi) 16-bit result
                          ; AUX, .Y unaffected
                          ; .Y can be negative
                          ;

:Mult8
         sty temp

                          ;dumb multiply
                          ;         sty acc
                          ;         lda #00
                          ;         tay
                          ;:mloop   clc
                          ;         adc sin0-$10,x
                          ;         bcc :skip
```

```
                          ;         iny
                          ;:skip    dec acc
                          ;         bne :mloop
                          ;         tya

         sty acc
         lda #00
         ldy #9
         clc
:mloop
         ror
         ror acc
         bcc :mul2
                          ;         clc
                          ;dec sin0 table by 1 instead
         adc sin0-$10,x
:mul2    dey
         bne :mloop

         ldy temp
         bpl :pos
         sec
         sbc sin0-$10,x
:pos

         sta (p4000),y
         eor #$ff
         clc
         adc #1

         STA (p6000),y
         iny
         bne :mult8
         inx
         inc p4000+1
         inc p6000+1
         bpl :mult8       ;to $8000

                          ;.x=$20
                          ;.y=0

*
* Set up Screen
*


                          ;         lda #00      ;bah!
                          ;         tya
:l2a
                          ;         pha
         jsr $e8ea        ;scroll up
                          ;         pla
                          ;.A = ($AC)
                          ;.X=00

         clc              ;necessary?
:l2                       ;set one line
         sta $0720+12,x
         inc $db20+12,x
         inx
         adc #16
         bcc :l2
                          ;         adc #$00
         inc $ac

         cmp #15
         bne :l2a
:done                     ;.X=$10

         stx $40          ;rats :(
                          ;init Py

*
* Main program loop:
*   - get input
*   - update positions/angles
*   - render
*   - swap buffers
*

MainLoop
```

```
                        ; Clear buffer and swap

          lda base
          eor #$08
          sta base

          sta zpoint+1
          ldy #00
          tya
          ldx #8
:l0       sta (zpoint),y
          iny
          bne :l0
          inc zpoint+1
          dex
          bne :l0

                        ; Get Input
                        ; .X=0

          lda $dc00
          lsr
          lsr
          lsr
          bcs :c1
          inc theta
:c1       lsr
          bcs :c2
          dec theta
:c2       and #$01
          sta tagged
          eor #$01
          ora $23c0+6
          sta $23c0+6       ;shot fired



                        ;         lda $cb
                        ;         and #3
                        ;         beq :skip
                        ;         lsr
                        ;         bcc :inc
                        ;         dec theta
                        ;         lsr
                        ;        bcc :skip
                        ;         jsr Forwards
                        ;:inc     inc theta
                        ;:skip

ObjLoop

                        ; Compute relative center

          inx
          stx curobj

          dec cenz,x        ;Update pos

          lda cenx,x
                        ;         sec
                        ;         sbc cenx
          sta Px+4
          lda cenz,x
                        ;         sbc cenz
          sta Pz+4


                        ; Rotate
Rot

          lda theta,x
          adc theta
          sta angle

          lda theta
          ldx #4
          ldy #STA
          jsr RotProj
          lda cz
          sbc #14
```

```
                        ;         bmi :skip
         cmp #125
         bcs NoDraw
         cmp cx
         bmi NoDraw
         adc cx
         bmi NoDraw

                        ; If in view, rotate project & plot

         dex
         stx count
:loop    lda angle
         ldy #ADC
         jsr RotProj
         sta YCoord,X
         dex
         bpl :loop

                        ; Plot

:ploop   ldy count
:l2      ldx count
         dey
         tya
         pha
         jsr DrawLine
         pla
         tay
         bne :l2
         dec count
         bne :ploop
                        ;.y=0
NoDraw
         ldx curobj

                        ; Update

         lda tagged,x
         beq :zip
         lda cx
         ora tagged
         sta tagged,x
         dfb $2c
:zip     inc theta,x


         cpx #NUMOBJS
         bcc ObjLoop

         jsr $eb59      ;Swap buffer
                        ;.Y=1 .A=$7F

         jmp MainLoop


*
* Line routine
*  - Draws from L to R
*  - Forces dx>0 (x1 < x2)
*  - Numbers are +128 offset, to allow
*  - DX and DY to be 0..255 for long lines
*  - Uses same core routine for stepinx
*    and stepiny by changing variables
*
* On input:
*  .X = index into point list, point 1
*  .Y = index of point 2
*

Pswap
         sty temp
         txa
         tay
         ldx temp
DrawLine
         lda xcoord,y
         sec
         sbc xcoord,x
         bcc Pswap      ;x2 >= x1
         sta dx
```

```
         lda ycoord,y
         sbc ycoord,x     ;dy = y2-y1
         bcs :posy
         ldy #DEY
         eor #$ff
         adc #1
         dfb $2c
:posy    ldy #INY
         sta dy
         cmp dx
         lda #INX
                          ;         bcs stepiny
         bcs :noswap
         tya
         ldy #INX
:noswap  sty mod5
         sta mod4
stepinx
                          ;         sta mod5
                          ;         sty mod4     ;INY/DEY
         lda dy
         ldy dx
         bcc mod
stepiny
                          ;         sta mod4
                          ;         sty mod5     ;iny/dey
         lda dx
         ldy dy
mod
                          ;         sty mod1+1
         sty temp
         sty mod3+1
         sta mod2+1

Draw
         lda ycoord,x
         sec
         sbc #64
         tay
         lda xcoord,x
         sec
         sbc #64
         tax
                          ;mod1     lda #dy
         lda temp
         beq done         ;no steps!
                          ;         sta temp
         lsr
plot
         pha
         tya
         bmi calcline
         lda bitphi,x
         ora base
         sta point+1
         lda bitplo,x
         sta point
         lda bitp,x
         beq calcline
         ora (point),y
         sta (point),y
calcline
         pla
mod2     sbc #dx
         bcs mod5
mod3     adc #dy
mod4     inx
mod5     iny
         dec temp
         bne plot
done
         rts

*
* Rotate and project point
*
* On entry:
*  .X = index into point list
*  .A = rotation angle
*  .Y = ADC/STA for points/centers
*  CX,CZ = location of object
```

```
*
* On exit:
*  Points stored in xcoord,x ycoord,x
*
RotProj
         AND #$3F
         ORA #>SINTAB
         STA sin+1
         SBC #$2F          ;-3*pi/4
                           ;         ADC #$10
                           ;         AND #$3F
         ORA #>SINTAB
         STA cos+1
         STY RotM1
         STY RotM2
RotProj2
         LDY Pz,X
         LDA (sin),Y
         PHA
         LDA (cos),Y
         LDY Px,X
         SEC
         SBC (sin),Y
         CLC
RotM1    ADC CZ
         STA aux

                           ;         LDY Px,X
                           ;         LDA (cos),Y
                           ;         LDY Pz,X
                           ;         CLC
                           ;         ADC (sin),Y
         PLA
         CLC
         ADC (cos),Y
         CLC
RotM2    ADC CX
         JSR Div8
         STA XCoord,X

         LDA Py,X
                           ;         JSR Div8
                           ;         STA YCoord,X
                           ;         RTS

*
* Signed division routine
*
* 64*.A/AUX -> .A
* Only valid for .Y/AUX < 1
* .A pos or neg, AUX > 0
* AUX, .X unaffected
*

DIV8
                           ;         STY ACC
                           ;         tya
         php
         bpl :pos          ;.A just loaded
         eor #$ff
:pos     sta acc
         LDA #0
         LDY #14
:DLOOP   ASL ACC

         ROL
         CMP AUX
         BCC :DIV2
         SBC AUX
         INC ACC
:DIV2
         DEY
         BNE :DLOOP
                           ;RTS

         lda acc
         plp
         bpl :pos2
         eor #$ff
:pos2
         eor #$80          ;+128 offset
```

```
         rts

*
* Point list
* Last point is for relative cx,cz
*

                            ;Pz       dfb 0,16,0,-16,0
                            ;Py       dfb 16,0,8,0
Px       dfb 0,-4,16,-4

                            ;note: px+4 intrudes on table below

*
* 128*sin(t), t=0..15
*
* Table must be at end of code!
*

                            ;sin0     dfb 0,13,25,37,49,60,71,81
                            ;         dfb 91,99,106,113
                            ;         dfb 118,122,126,127,128

                            ;sin0     dfb 0,25,50,74,98,120,142,162
                            ;         dfb 180,197,212,225,236,244,250,254,255
sin0     dfb 0,24,49,73,97,119,141,161
         dfb 179,196,211,224,235,243,249,253,255


end

.......
....
..
.                                    - fin -
```