Technical details of Musik Run/Stop

Playing 8-bit samples at 44.1kHz on a computer from 1982.

Author: Pex 'Mahoney' Tufvesson, M.Sc.EE
Lund, Sweden, February 2014

This white paper is written to help you understand how the digital and analog audio processing was implemented in the Commodore 64 demo "Musik Run/Stop" written by Pex 'Mahoney' Tufvesson in February 2014.

To fully enjoy this paper, some knowledge about digital audio and computer programming is required. You do not really need knowledge of 6502 assembly programming or Commodore 64 hardware programming.

Keywords: 8-bit Digital Audio, Commodore 64, 6502 CPU, extreme optimizations

I. Introduction

This white paper will describe how to manage to play 8-bit samples at 44.1kHz with a home computer from 1982.

II. System Requirements

A. *Commodore 64*

- 64kB RAM
- 6502 processor, 8-bit, 1MHz
- RF antenna output, PAL video standard
- 6581/8580 SID sound chip, 3 oscillators, 1 filter

B. *TV*

C. *A brain. Yours, preferrably.*

III. Acknowledgements

Huge thanks to Uwe "THCM" Anfang for helping out with measurements on real hw. Also thanks to Johan "Bepp/Triad" Book and "Hedning/G*P" for making a stack of C64 computers available for this research.

IV. Screen Shots

Figure 1. Screen Shots

Think of all the things you can do while not watching TV

V. Video Shot

There are a number of versions of "Musik Run/Stop" uploaded to YouTube. The funniest is the live version with the audience reactions from the release party on the 15th of February: Datastorm 2014 in Gothenburg, Sweden. Go search for "musik run/stop mahoney" at youtube.com and you'll find them all.

If you're not happy with a video version of it, go and grab an emulator of the Commodore 64. There are many, but I'd recommend the vice emulator at http://www.viceteam.org

With any modern computer and operating system, the Commodore 64 can be emulated quite ok. You'll have to cope with the monitor refresh being out of sync with the PAL TV's 50Hz video, though. And this time, the audio SID emulation isn't good enough – whatever SID software emulation you'll try, it falls back to approximately 5-bit audio output resolution, which really isn't good enough. At least not for me. The fallback is to enable an additional emulated digital output cartridge called "Digimax" and press a key when the demo starts, this will make all of the demo output pure 8-bit samples with a perfect linearity. If you're happy with that, then you can stop reading this document, since the rest of the text here will describe to you what really happens "under the hood" of the 8-bit samples as produced by the SID 6581/8580 sound chip.

If you're the happy owner of the real deal, please run Musik Run/Stop on your Commodore 64. It's been running ok from real floppes, but also using an "Ultimate II" USB-memory/sd-card-based 1541 emulator. If you don't own a Commodore 64, it's a flea-market bargain at an approximate price of 500 SEK, €50, $50 or something similar. Oh! The joys of cheap retro computing!

VI. Specification

This is a list of all the requirements for Musik Run/Stop:

- Use a Commodore 64 home computer with a CPU constructed back in 1975, that's 39 years ago. The 6502 CPU used about 4000 transistors, compared to a recent 2010 Intel CPU with 1.17 billion transistors. The SID sound chip was designed in 1981, and is described in the U.S. Patent 4,677,890, which was filed on February 27, 1983, and issued on July 7, 1987. The patent expired on July 7, 2004.
- The CPU runs at 985.248 Hz, which is less than 1MHz.
- We want to play samples at approximately 44.1kHz

VII. How much time do we have?

The 6502 central processing unit (CPU) in the Commodore 64 computer is run at almost 1MHz. This means that there's a clock ticking 985.248 times per second.

Thankfully, there are support chips in the Commodore 64 that will help us. But just a little. The 6526 CIA peripheral chip is used to interrupt the 6502 CPU every time we might want to trigger a new sample. But for playing samples this fast, these timing chips are useless.

So we have 985248 CPU clocks per second, and we want to output 44100 samples per second. That's 985248/44100 = 22.3 clock cycles available per output. Let's choose 22 clock cycles as target length of our output loop, and we'll end up with 985248/22 = 44784 Hz, which is good enough.

An assembly instruction on the 6502 CPU takes between 2 and 8 clock cycles to run. 22 clock cycles will approximately be 6 assembly instructions, which is all we have available per sample.

So, the answer to the question "How much time do we have?" is:

6 assembly instructions for calculating every new 8-bit sample.

VIII. How much memory have we got?

A first-approach sample player on Commodore 64 will pick 1 byte at a time from memory, and output that at a constant rate to the SID chip. If the sample rate is 44784 Hz, filling the whole memory with nothing but sample data will get us

64kB / 44784 = 1.46 seconds of nice music.

This is not much. But, just face the facts: with 6 assembly instructions per sample, there's very little room for adding sample decompression at run-time.

And, to make things worse, we do not have 64kB available to store samples in. The VIC-II registers, SID registers, color memory, and CIA registers do have useful memory "underneath them", but unfortunately, we don't have the CPU time to switch these off, read a value, and then switch them on again. We need that CPU time for calculations. We lost 4kB. Then, we need to keep the resident part of the disc loader in memory, which is another 1290 bytes.

But, let's skip any kind of screen memory. And only use the colour memory for the graphics. And two sprites (128 bytes) for outputting all the graphics. And "hide" the full screen images underneath the VIC-II, SID, CIA and color memory.

The 6502 stack (which I've reduced to just 48 bytes), and the 6502 zero page registers will occupy 304 bytes. We're down to 65536-4096-1290-128-304 = 59718 bytes of memory. And, this memory needs to be shared between the actual program playing the sound, the sound data itself, and a loader routine that can read the next part of the demo after the 44.1kHz sample playing is done.

IX. Memory Map

It's good to have a plan. So, the complete memory map now looks like this:

$0000-	Zero page registers and self-modifying code
$0100-	6502 CPU stack
$0140-	Resident part of disc loader
$0700-	Sample memory
$d000-	32 full screen graphics bitmaps (25*40 pixels=125 bytes each)
$e000-	Audio effects code, sequencer tables
$f400-$fff7	Loader music when loading the next part

Figure 5. Memory layout

X. This must have been done before

If we have 6 assembly instructions per sample, then all the current known 8-bit sample playing tricks on the Commodore 64 fails:

There is the "one voice, test-bit, triangle-wave, sample-and-hold by just briefly enabling waveform"-technique invented by Otto "SounDemoN" Järvinen in 2008. It has been used properly in 16kHz, but with very low output amplitude and around 5-6 bits resolution. Used at 8kHz, the amplitude is higher – and actually quite enjoyable. But nowhere near 44.1kHz sample rate. The main reason is that it requires 4 writes to the SID chip for every output sample (well, three if you cheat and don't mind having a 16kHz carrier frequency). So, setting up a sample would require at least 24 clock cycles for the SID handling only – and the amplitude would be ridicuously low anyway.

There is the "pulse-width"-technique for playing samples, but that has even worse carrier noise, and would not be able to get any higher frequiencies than the highest pitch the SID chip can produce, which is around 4kHz.

And we have the 4-bit "write to the volume register of the SID", which doesn't work well on newer SID chip revisions, since the 8580 SID chip removed the internal offset voltages for the three voice channels. And, it's only 4 bits. But it has one very big advantage: it needs a single write to the SID chip, and it's truly jitter insensitive – you don't need any fancy CPU timing to get the amplitude right.

So, there's no sample playing technique that will do the trick of playing 44.1kHz samples. We need to think. And we need to invent something new.

XI. What registers do we have?

This is the register map as taken from the original SID 6581 data sheet:

SID CONTROL REGISTERS

There are 29 eight-bit registers in SID which control the generation of sound. These registers are either WRITE-only or READ-only and are listed below in Table 1.

Address A4	A3	A2	A1	A0	Reg # (Hex)	Data D7	D6	D5	D4	D3	D2	D1	D0	Reg Name	Reg Type
														VOICE 1	
0	0	0	0	0	00	F7	F6	F5	F4	F3	F2	F1	F0	Freq Lo	Write-only
1	0	0	0	0	01	F15	F14	F13	F12	F11	F10	F9	F8	Freq Hi	Write-only
2	0	0	0	1	02	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	PW LO	Write-only
3	0	0	0	1	03	—	—	—	—	PW11	PW10	PW9	PW8	PW HI	Write-only
4	0	0	1	0	04	NOISE	⊓⊓	◿◺	∧∧	TEST	RING MOD	SYNC	GATE	Control Reg	Write-only
5	0	0	1	0	05	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/Decay	Write-only
6	0	0	1	1	06	STN3	STN2	STN1	STN0	RIS3	RIS2	RIS1	RIS0	Sustain/Release	Write-only
														VOICE 2	
7	0	0	1	1	07	F7	F6	F5	F4	F3	F2	F1	F0	Freq LO	Write-only
8	0	1	0	0	08	F15	F14	F13	F12	F11	F10	F9	F8	Freq Hi	Write-only
9	0	1	0	0	09	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	PW LO	Write-only
10	0	1	0	1	0A	—	—	—	—	PW11	PW10	PW9	PW8	PW HI	Write-only
11	0	1	0	1	0B	NOISE	⊓⊓	◿◺	∧∧	TEST	RING MOD	SYNC	GATE	Control Reg	Write-only
12	0	1	1	0	0C	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/Decay	Write-only
13	0	1	1	0	0D	STN3	STN2	STN1	STN0	RIS3	RIS2	RIS1	RIS0	Sustain/Release	Write-only
														VOICE 3	
14	0	1	1	1	0E	F7	F6	F5	F4	F3	F2	F1	F0	Freq Lo	Write-only
15	0	1	1	1	0F	F15	F14	F13	F12	F11	F10	F9	F8	Freq Hi	Write-only
16	1	0	0	0	10	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	PW LO	Write-only
17	1	0	0	0	11	—	—	—	—	PW11	PW10	PW9	PW8	PW HI	Write-only
18	1	0	0	1	12	NOISE	⊓⊓	◿◺	∧∧	TEST	RING MOD	SYNC	GATE	Control Reg	Write-only
19	1	0	0	1	13	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/Decay	Write-only
20	1	0	1	0	14	STN3	STN2	STN1	STN0	RIS3	RIS2	RIS1	RIS0	Sustain/Release	Write-only
														Filter	
21	1	0	1	0	15	—	—	—	—	—	FC2	FC1	FC0	FC LO	Write-only
22	1	0	1	1	16	FC10	FC9	FC8	FC7	FC6	FC5	FC4	FC3	FC HI	Write-only
23	1	0	1	1	17	RES3	RES2	RES1	RES0	Filt EX	Filt 3	Filt 2	Filt 1	RES/Filt	Write-only
24	1	1	0	0	18	3 OFF	HP	BP	LP	VOL3	VOL2	VOL1	VOL0	Mode/Vol	Write-only
														Misc	
25	1	1	0	0	19	PX7	PX6	PX5	PX4	PX3	PX2	PX1	PX0	POTX	Read-only
26	1	1	0	1	1A	PY7	PY6	PY5	PY4	PY3	PY2	PY1	PY0	POTY	Read-only
27	1	1	0	1	1B	07	06	05	04	03	02	01	00	OSC3/Random	Read-only
28	1	1	1	0	1C	E7	E6	E5	E4	E3	E2	E1	E0	ENV3	Read-only

TABLE 1 — SID REGISTER MAP

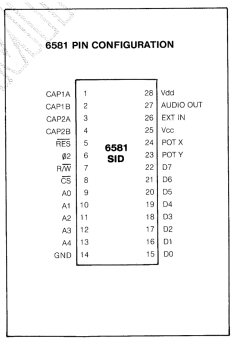
If we only have 6 assembly instructions per sample available, then we will need to find *one* single write that will output our sample.

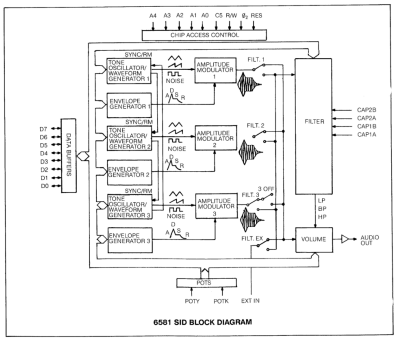
We can safely ignore anything related to VOICE1,2 and 3 in the table above, there just isn't anything in there to exploit. Under the "Filter" section in row 23, there is a 12-bit filter cutoff value, and a resonance and filter voice enable bits, which is quite interesting. And at row 24 is the highly interesting Mode/Volume register, which controls "3 OFF", HP, BP, LP and a 4-bit volume value.

The register under "Misc" are read only, so there's no luck in using them at all.

XII. Any kind of block diagram would be nice

Again, taken from the SID 6581 original datasheet:





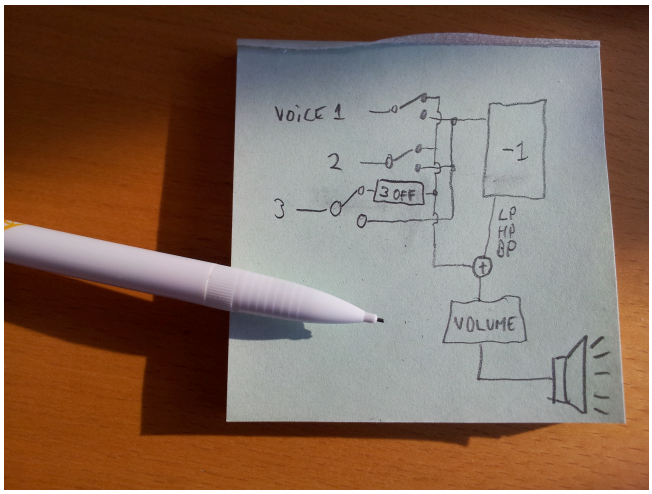
XIII. Let's start thinking

One year ago, I made a song called "Monophono" – which accidentially won the C64 music competition at Datastorm 2013. When programming that song, I had huge problems trying to get rid of a couple of loud clicks that occurred in the song. I know the SID chip needs special love when choosing how and when to setup the registers, but these clicks were in the middle of the song.

I realised that the clicks came whenever I toggled a voice from being routed through the analog filter of the SID chip, or not. And, the clicks came louder when doing it on a voice channel that currently was playing a waveform.

How come? I went back to my own software SID emulator (similar to reSID, but my own version of it), and started to read my own code. Nothing special there that could explain the clicks. So, I went to Bepp/Triad's place to do some measurements on real hardware. Yes, the clicks are there. And looking at the audio output, I realised that the amplification factor of the SID chip's analog filters wasn't 1, it was actually around -1. Now, you have to realise that most waveforms that the SID chip produces are very symmetric: triangle waveforms, pulse waveforms and noise will look identical when turned "upside-down". But, fortunately, the sawtooth waveform will not!

Armed with this new knowledge, and with an updated SID software emulator, it was time to grab a pen and paper and make some drawings of the innermost parts of the SID chip:



We do have luck on our side: the three leftmost muxes are normally controlled by register 23 (Filt 1,2,3) – but partly also by register 24! If the Mode/Vol register is written so that no filtering is enabled (HP=LP=BP=0), then the muxes will pass the signal to the non-filtered bus as well.

Another lucky thing is that the "3 OFF" bit only shuts off voice 3 when it's not filtered. Normally, this bit was intended to silence voice 3 when it was used for random number generation (selecting the noise waveform and reading OSC 3 output through register 27.

And, the last lucky thing is that the analog filter has an amplification of "almost -1" – but nobody is perfect, and let's use those imperfections to our advantage.

XIV. So how do we setup the SID chip for magic?

```
lda #$0f  ;Setup attack=0 and decay=15
sta $d405
sta $d40c
sta $d413
lda #$ff  ;Setup all sustain&release to 15
sta $d406
sta $d40d
sta $d414
lda #$49  ;Waveform is square, test bit set
sta $d404
sta $d40b
sta $d412
lda #$ff  ;Filter cutoff as high as possible
sta $d415
sta $d416
lda #$03  ;Enable voice 1 and 2 through filter
sta $d417
```

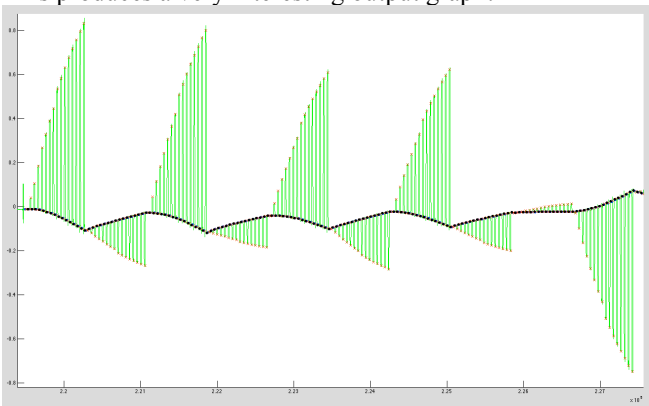
With this setup, all voices will "unfiltered" output their maximum values. The envelope generators will also be set to give maximum gain to each voice. For 6581 let's call this "1 + internal offset voltage".

XV. Time to start measuring one SID chip

With this setup, let's run a short measurement program on the real thing, and measure the audio output. The short version of the program is:

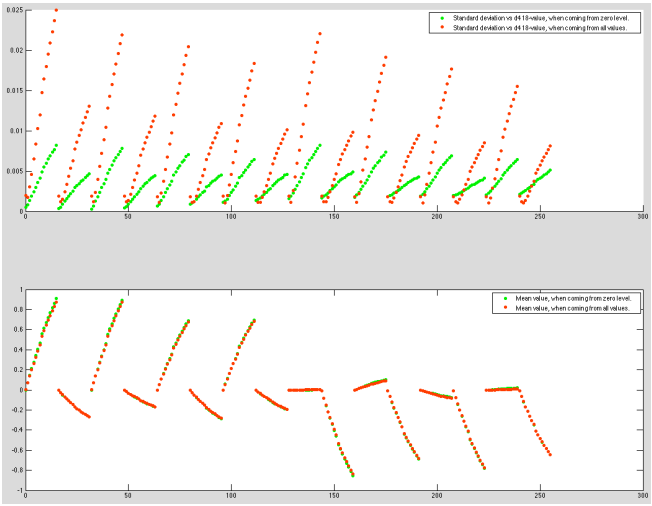
```
  X=0
another2:
  Y=0
another:
  write #0 to $d418
  write  X to $d418
  write  Y to $d418
  increment Y
  as long as Y<=255, do another
  increment X
  as long as X<=255, do another2
```

This produces a very interesting output graph:



This shows the first ~160 values of Y in the program above. We can see the "black line" jumping around a little, and this is due to the 16Hz high-pass filter on the SID chip output. We don't have a static zero-level reference.

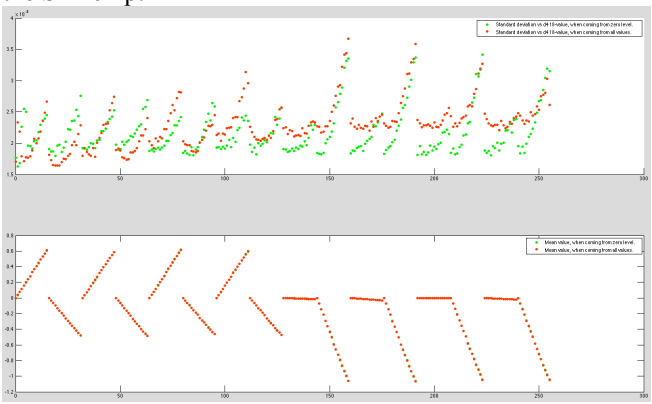
But, we do have multiple measurements of the same $d418 write, so for one 6581 SID chip, we get this:



The upper graph shows the standard deviation within one chip. The red dots are how much the output amplitude deviates when coming from all other values. And the green dots shows how much the output amplitude deviates when coming from the zero level.

The lower graph shows the average output level produced for all 256 possible values written to $d418. This isn't the normal 4-bit sample output we used to get when writing to the volume register. This looks promising.

Let's do the same thing with the 8580 "new revision" of the SID chip:



From the upper graph we can see that the standard deviation is way smaller (up to 0.004, compared to the 6581's 0.025). It is a "better" chip, but in this case it's actually worse. It is more stable, that's true – but looking at the lower graph, we have lost most of the analog imperfections. Still, it's an interesting find.

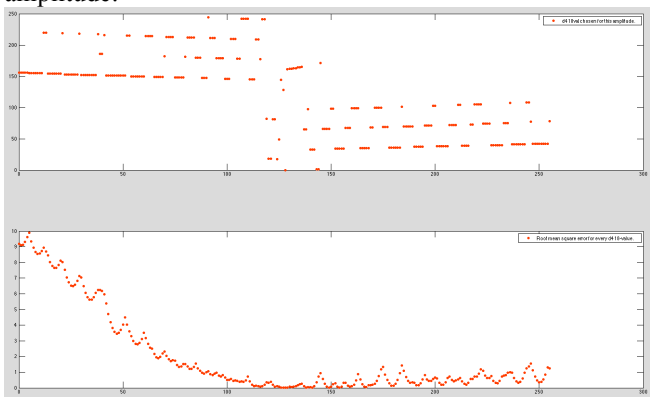
XVI. TIME TO MEASURE MANY SID CHIPS

In order to give any reasonable assurance that this might work on more than one single SID chip, we measured more chips, and we could start making statistics out of them.

And out of this, we got a translation table telling us "if you want this output amplitude, then please write value X into $d418". It has got 256 different values – but – they are not linearly distributed at all. But that doesn't matter much, does it?

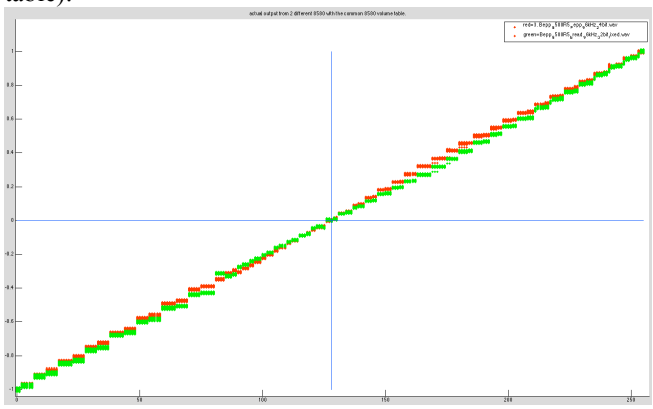
Here's what writing these values will get you – showing all single measurements on all 6581 chips:

We can analyze "how good linearity" we have, by calculating the standard deviation for every single amplitude:

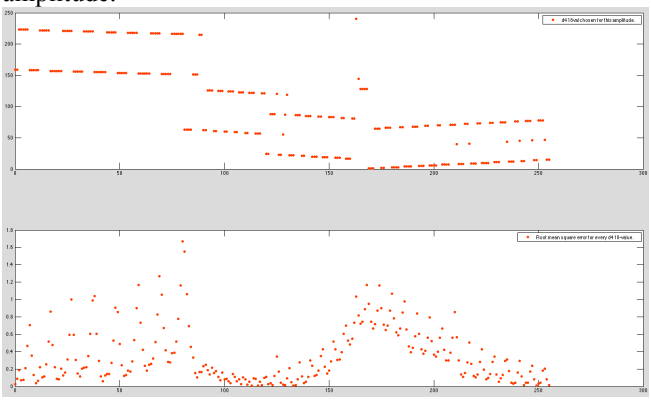


In the graph above, we see every single "chosen" $d418 write to represent the amplitude, and in the lower half we see the standard deviation from the desired amplitude. It shows how much we deviate from a straight line.

And for all 8580 chips (with the common 8580 amplitude table):



We can analyze "how good linearity" we have, by calculating the standard deviation for every single amplitude:



In the graph above, we see every single "chosen" $d418 write to represent the amplitude, and in the lower half we see the standard deviation from the desired amplitude. It shows how much we deviate from a straight line.

XVII. THEN HOW DO WE DO IT?

Well, we need to find those 6 assembly instructions that can address the memory and write the sample into the SID $d418-register. The innermost loop could look something like this:

```
another:
  lda sample,x  ;Load sample value
  sta $d418     ;Send to SID register
  inx           ;increase x
  bne another   ;and go get another one
```

...that is four assembly instructions per sample, actually taking only 4+4+2+3 = 13 clock cycles. Which is 985278/13 = 76kHz audio.

But the problem is that this loop will only handle 256 bytes – then we need to add additional instructions to keep reading more data from the next page (a group of 256 bytes in memory). And now we need to introduce self modifying code, which is nothing to be afraid of in a non-cached flat memory architecture like the Commodore 64:

```
another:
current_pointer:
  lda sample,x  ;Load sample value
  sta $d418     ;Send to SID register
  inx           ;increase x
  bne another   ;and go get another one
  inc current_pointer+2  ;increase page number
  jmp another   ;and go get the next page
```

This might look like a viable alternative – but this loop will take different number of clock cycles every time we need to increase the high bits of the current_pointer: 13 clock cycles for 255 samples, and then 13+5+3 = 21 clock cycles for the 256th sample. This would sound terrible, and what's worse – the sample would never stop – playing all of the memory in one continuous loop.

So, we need to waste some clock cycles inside this loop just for fun (remember, we were playing samples at ~76kHz in the innermost loop, and that's a little bit too fast, I think!). To help with the visual part of the demo, we could for instance send the sample value not only to the SID chip, but to the VIC-II chip as well – as a colour value for the screen.

Main problem with this is that the C64 VIC-II chip uses 16 colours, and they are heavily "unsorted". So, to be able to turn something like "output amplitude" into a luminance value, we'll need a table. So, here we go:

```
another:
current_pointer:
  ldy sample,x  ;Load sample value
  lda colourtable,y
  sta $d020
  sty $d418     ;Send to SID register
  inx           ;increase x
  bne another   ;and go get another one
  inc current_pointer+2  ;increase page number
  jmp another   ;and go get the next page
```

Now the innermost loop takes 4+4+4+4+2+3=21 clock cycles. Which is good. A little bit too fast (~47kHz), but that can be fixed. Still, when going to the next page, we have 21+5+3=29 clock cycles in the loop. Which we'll need to fix somehow.

So, we will need to be able to grab every "page-breaking" sample value a little faster than the others. What about this version, then:

```
  ldx #0
another:
current_pointer:
  ldy sample,x  ;Load sample value
  lda colourtable,y
  sta $d020
  sty $d418     ;Send to SID register
  inx           ;increase x
  bne another   ;and go get another one
  inc current_pointer+2  ;increase page number
  ldy current_pointer+2
  lda sample_table,y
  sta $d418
  bne another   ;and go get the next page
  rts           ;we're done playing this sound
```

Well, now we need to know what's important. The most important feature of this program is that the distance in clock cycles between every write to the $d418 register is constant. The innermost loop was 21 clock cycles long. So, we need to check the number of clock cycles between the "st_ $d418"-instructions.

From the inner-loop sty $d418 to the outer-loop sta $d418 there is 2+3+5+4+4+4=22 clock cycles. And from the "sta $d418" to "sty $d418" there is 3+4+4+4+4+2 = 21 clock cycles. We're getting closer.

So, we need to waste one clock cycle somewhere inside the inner loop, which is harder that it sounds like. The main problem is that the normal assembly instruction for wasting clock cycles the no-operation "NOP" actually wastes not one but two clock cycles. So, it's useless. We'll need to find another way of doing nothing.

16-bit arithmetics in an 8-bit CPU is the solution. If we do place the "colour_table" at a bad place in memory, then the CPU will have to propagate a carry signal to the upper 8

bits of the address bus when adding the y register to the memory address. The natural place in memory for a colour_table would be aligned to a page in memory (addresses looking like $xx00 – ending with two zeroes). But, if we align this table to $xxff, then whenever y is anything else than 0, the CPU would waste one clock cycle increasing the uppermost 8 bits of the address bus. So – job done, without changing anything in the code.

As an additional bonus – a zero value in the sample data at every 257th sample will break this loop and stop playing. Which is good, we found a way of ending sample play as well.

The code will use exactly the same number of clock cycles, as long as the sample values in the inner loop are anything else than 0.

XVIII. JOB DONE

This is how we do it. And by all means, you're welcome to use the knowledge you gained by reading this white paper however you want. You can grab the files mentioned in this white paper somewhere at http://livet.se/mahoney

For more information about the Commodore 64 “sceners”, people still programming audio-visual entertainment on this old home computer, see the Commodore 64 scene database, http://csdb.dk

XIX. DISCLAIMER

There's probably loads of errors in this text. If you did find one, I'd be happy if I got to know about it. You'll find my contact details at my homepage

http://mahoney.c64.org

Some of the "errors" are deliberate, since telling the whole truth and nothing but the truth would miss the educational flow of the text. If you want the full Monty on the run, please read the enclosed source code of the measurement program and the Matlab amplitude table extraction tool. It's all there. Unabridged. And probably completely incomprehensible to 99.9% of the human population.

Most of the comments found in the source code are meaningful. But there are traces of work-in-progress comments that should have been cleaned up long ago. The source code is correct, and most of the comments are too. If you find your way around the code, it's a fun read. But, it is not for the faint of heart, and please mind the gap.

Thanks for reading. I hope you have learnt something new and I hope you feel it was time well spent. Please stop by my homepage http://mahoney.c64.org and give me a comment or two.

Best Regards, Pex 'Mahoney' Tufvesson, Lund, Sweden, February 2014.

Pex 'Mahoney' Tufvesson, M.Sc.EE., has been programming computers since 1979. PET, ABC-80, Sinclair ZX Spectrum, Commodore 64, Amiga 500, Nintendo 64, Mac and PC. He's currently working as a hardware engineer, creating chip designs, and is the webmaster of a couple of websites like http://www.livet.se/ord which is a proverb collection. He's a musician with his own a cappella group http://www.livet.se/visa Visa Röster. You'll find more about him on his homepage http://mahoney.c64.org