Vicious Sid Demo Routine Explained

Discussion about PWM and samples that are referred at the beginning of the article:

- http://www.ffd2.com/fridge/chacking/c=hacking20.txt “The C64 Digi”
- http://www.ffd2.com/fridge/chacking/c=hacking21.txt “Pulse Width Modulation, continued”

Vicious Routine

In this article I will describe a new kind of sample player routine which can play crystal clear 8 bit samples! Even with the screen turned on. This is the routine that was used in our demo Vicious Sid from X2008. It is based on the idea of using SID waveforms to construct the samples.

Okay, you can't call 8bit,8kHz crystal clear, but at least this method sounds much better than playing samples using the volume register ($D418) or the method using pulse width modulation. The problem with $D418 samples is that they're limited to 4 bits. With pulse width modulation (PWM) you can achieve 12 bit samples but instead you will hear a very loud and ear-piercing 4khz tone together with the played sample. There are other less known PWM based methods that produce pretty good quality samples, like the one by Ninja in Darwin. I believe this is the same method or similar to the method invented by the Levente Harsfalvi presented in C=Hacking #21. The drawback is that these routines cost you all the raster time.

The Problem

The idea for this kind of routine isn't completely new. I know that at least Krill and Agemixer have been planning on using fragments of SID waveforms for creating more complex waveforms after on IRC discussion on #c-64 with Exin and Necronomfive about the possibility to play samples by utilising SID waveforms I came up with a new idea.

As I see it, the biggest problem with creating more complex waveforms by utilising the existing SID waveforms is that you'd want to quickly jump to specific positions in the SID waveforms. The SID doesn't have a mechanism that allows this. The only way is all the way through the whole waveform. It takes circa 256 cycles at frequency $ffff to play one period of a waveform, which is simply not fast enough.

For example, one idea would be to use the saw waveform to set the desired DAC level for each sample. We could do this by resetting the oscillator of a SID voice with the test bit. Then we could run it until it reaches the desired level and stop it by setting its frequency to $0000. However, the SID is too slow for this purpose.

It would now be easy to conclude that playing samples by using SID waveforms isn't possible; the oscillators are too slow, and we're stuck at a specific position in a waveform too long.

However, usually these kinds of statements contain some untrue assumptions. It's the beauty of C64 coding to do seemingly impossible things.

I will start by presenting an implementation that is easy to understand and easy to implement. I will then present a more advanced method.

The First Implementation

The problem was to overcome the slowness of the SID oscillators. Surely it's not possible to overclock the SID with software…But who said we are limited to only once? My solution was to use two oscillators instead of one. It might be impossible to play samples with only one voice, but what about two voices? The trick is to have one voice sounding while the other one is “seeking” the right level for the next sample.

Just like in the example with the saw waveform mentioned earlier, but with two voices. When it is time to play the next sample, the roles of the voices are swapped. This way the transition from the current sample level to the next doesn't have to fast, and oscillator slowness isn't a problem.

By seeking I mean running the oscillator until it reaches the desired sample level. The oscillator of a SID voice is a 24bit register, to which the frequency value is added every cycle. This register is called the phase accumulator. All waveforms (expect noise) are generated by the help of the value in the phase accumulator. The waveform is generated by outputting the value of the phase accumulator itself. This means that we can send the value in the phase accumulator to the DAC (“play it”) by just toggling on the saw waveform.

Seeking is accomplished like this:

1) Clear the value in the phase accumulator by setting the test bit
2) Set the FREQUENCY of the oscillator to the sample level we wish to seek
3) Start the oscillator (clear the test bit) and disable output (waveform #00)
4) Wait a CONSTANT amount of time (typically two raster lines)

As already mentioned, the frequency value is added to the phase accumulator each cycle. So after N cycles the value in the phase accumulator will be N*frequency. In other words, throwing the desired sample level to the frequency register actually works (step2), and the longer we wait (step 4) the louder the playback will be. The delay in step 4 has to be to 256 cycles or less. Otherwise the value in the phase accumulator might wrap.

Now that the phase accumulator contains the desired sample level it can be “played”:

5) Stop the oscillator by setting its frequency to 0
6) Select the saw waveform (triangle waveform works as well)

If the delay in step 4 is 128 cycles or less, it's usually better to use the triangle waveform instead of the saw waveform. The triangle waveform is similar to the saw waveform, but rises twice as fast. Hence using the triangle waveform means doubling the volume. The maximal delay in step4 is then 128 cycles, since the latter half of the triangle waveform is unusable.

So code doing this with two voices could look something like this:

```
IRQ1:
STA $FE
```

```
STY $FF

LDA #$00 ;step 5(voice 3)
STA $D40F
LDA #$11 ;step 6(voice 3)
STA $D412

LDA #$09 ;step 1(voice 2)
STA $D40B
LDY #$00 ;step 2(voice 2)
LDA ($10),Y
STA $D408
LDA #$00 ;step 3(voice 2)
STA $D40B

;step 4 until next irq (voice 2)

INC $10
BNE *+4
INC $11
LDA $DC0D

LDA #<IRQ2
STA $FFFE
LDA #>IRQ2
STA $FFFF

LDA $FE
LDY $FF
RTI

IRQ2:
STA $FE
STY $FF

LDA #$00 ;step 5(voice 2)
STA $D408
LDA #$11 ;step 6(voice 2)
STA $D40B

LDA #$09 ;step 1(voice 3)
STA $D412
LDY #$00 ;step 2(voice 3)
LDA ($10),Y
STA $D40F
LDA #$00 ;step 3(voice 3)
STA $D412

;step 4 until next irq (voice 3)

INC $10
```

```
BNE *+4
INC $11
LDA $DC0D

LDA #<IRQ1
STA $FFFE
LDA #>IRQ1
STA $FFFF

LDA $FE
LDY $FF
RTI
```

As you can see, there's no waiting (step 4) inside the IRQ. The waiting happens outside the IRQ, in order to save raster time. This was suggested by Mixer. At first I was against it because having an unstable raster would result in a delay that would vary. I thought this would add a high pitched carrier noise to the sound. However, this doesn't happen. The reason will be explained later. The beauty of this implementation is indeed that a stable raster isn't required.

The Second Implementation

This version is a bit trickier due to raster timing. A stable raster is required. The idea is pretty much the same as for the first implementation.

So I stated that it might be impossible (what would Crossbow do) to play samples by using just one voice due to oscillator slowness. Ironically, it turns out that even the assumption about two channels being required is false. This is because it's actually possible to play AND seek simultaneously on a single SID voice. Only Amig... C64 makes it possible.

How can this be possible? It's done by using no waveforms at all! In the first implementation we used waveform $00 to “mute” a voice. Initially I thought this would result in a zero level output from the DAC. Later I found out what really happens is that the current waveform level is sustained (the effect lasts a few seconds, after this the DAC level will decay to 0).

Knowing this, we can use the following method:

1) Start playing a sample (enable saw/triangle)
2) Sustain the current DAC level (disable saw/triangle output)
3) Reset phase accu (set test bit)
4) Set the FREQUENCY of the oscillator to the sample level we wish to seek
5) Start seeking (clear test bit) and wait...

And in code:

```
IRQ STA $FF

LDA $DC04
EOR #$07
```

```
AND #$07
STA *+4
BPL *
CMP #$C9
CMP #$C9
CMP #$24
NOP

LDA #$11 ;step 1
STA $D412

LDA #$09 ;step 2 & 3
STA $D412

IRQ1 LDA $1000 ;step 4
STA $D40F

LDA #$01 ;step 5
STA $D412

INC IRQ1+1
BNE IRQ2
INC IRQ1+2

IRQ2 LDA $DC0D
LDA $FF
RTI
```

This method is much trickier to implement and hence I recommend starting with the two voice version. It should b easier to modify a two voice version to a one voice version than writing it from scratch. The timing is very crucial and the number of cycles is very limited on badlines. Try inserting a few instructions in the above routine and it won't work anymore.

The LDA $DC04 thing in the beginning of the IRQ routine makes the raster stable. The idea is to read the current position in the rasterline from $DC04 and synch according to that. I'm not going to describe how it works in detail here. There's also a faster but more complicated way to do the stable raster code with the CIA. An example on how to do it can be found at:
http://www.dekadence64.org/prodstools.html

Examining Waveform $00

You can use this little program to watch the effect of waveform $00. It will first set the voice output to $ff and then toggle waveform $00. As can be seen from the screen, the output of the voice will “fade” from $ff to $00 within some seconds.

```
LDA #$00
STA $D410
STA $D411
```

```
LDA #$40
STA $D412
LDA #$00
STA $D412

LOOP LDA $D41B
STA $0400
JMP LOOP
```

You might wonder why you don't get a very noticeable carrier with the two voice version, even though your raster isn't stable. The running time ofd the oscillator, and as a result, the amplitude will vary which indeed DOES produce carrier noise.

Since the two version “mutes” the channel by setting waveform $00, the effect is that the current and the previous samples are played simultaneously. The averaging of the samples will smoothen the waveform and remove sharp transitions. As a result the played sample gets low pass filtered and the high pitched carrier noise disappears. Remember that high frequencies are represented by sharp transitions in the sample data. This also means that the one voice version sounds clearer since it's not low pass filtered.

To my knowledge the only emulator supporting the sustaining behaviour of waveform $00 is Hoxs64. I also know that Antti Lankila added this behaviour to his patched ReSID version. On other emulators both the 1 and 2 voice versions will sound very poor.

Conclusion

The disadvantage of this method is of course that you lose at least one SID voice. But in exchange for that you get clearer sample replay than with other methods. Another advantage compared to $D418 samples is that the method works on both 6581 and 8580.

Since the samples are generated with SID voices, it's also possible to filter the voice(s) or to do volume tricks with the ADSR. Mixer did a good job of demonstrating this in Vicious Sid.

It should also be possible to make a 12 bit routine, although I'm not sure it would be worth it due to the lost memory and added complexity.

The waveform DAC of the SID is quite non-linear. In order to improve the sound quality this nonlinearity would need to be examined. I suspect the non-linearity can vary quite a lot between various SIDs, so compensating for it might not be trivial (remember how some 6581 SIDs sound saturated even without any filtering, while others sound clean).

I'm providing some example code for your pleasure. SInce I'm suspecting almost no one's using TASS these days, I won't bother including sources on this disk. Instead you can download some example sources in text format from (not yet there, will be soonish):
http://www.dekadence64.org/prodstools.html

As a punishment for your cross compiler lameness the sources will be in traditional C64 TASS format. ;) You will probably have to make some small changes in order to adapt them to your assembler.

Happy hacking,
SounDemon.