Basic guide for creating sid music with goattracker V2.67
---

 ---
|  This guide is aimed at musicians that want to start creating .sid  |
|  chiptunes from scratch using the pc/mac application "goattracker".  |
 ---

___

__/Contents:
---



 ___
/  \

## Chapter 0 - Getting to know binary and hexadecimal numbering system

If you already know how to *think* in binary and hex, and know what a signed hex value is, you can skip to chapter 1.

___

__/BIT & Nybble
---

Humans daily life is dealing with decimal numbers, which we all know. Since computers work somewhat different from humans, and only know "charge" (1) or "no charge" (0) the concept of binary values has been made up.

One single "charge"/"no charge" value is known as bit. If we put together four of these values, we have a nybble. Binary numbers are indicated by a % in front. The lowest value a nybble can hold is %0000, the highest would be %1111. Going through all possible 0 and 1 combinations, we see that there are 16 different combinations possible.

Binary system is built up on the powers of 2, where the lowest bit represents a decimal value of 1. So the bits from LSB to MSB are 1  2  4  8 (and 16  32  64  128 when it's two nybbles/a byte).

A bit set to "1" means, we need to add it's decimal value up to the sum to know the final value of the nybble. Mind that the lowest bit is always referred to as bit 0. People also use the orientation LSB (least significant bit = bit 0) and MSB (most significant bit).

```
                MSB            LSB
bit number       3    2    1    0
dec value        8    4    2    1
                 :    :    :    :
                 :    :    :    :
example          :    :    :    :
bin value       %1    0    1    0  =  decimal 10 (adding up bit 1 and 3)
```

---

## __/HEX

HEXadecimal means that the number systems has a base of 16 (instead of 10 as we are used from decimal numbering). this was the most logical choice, since a nybble can hold a value between dec 0-15.

As decimal system only has 10 numerics for numbers, the letters A-F were incorporated into the system. After the value exceeds 9, it goes from A to F.

Hex numbers are indicated by a $, for better determination.

### Simple conversion table:

| Dec | Bin | Hex |
|---|---|---|
| 0 | %0000 | $0 |
| 1 | %0001 | $1 |
| 2 | %0010 | $2 |
| 3 | %0011 | $3 |
| 4 | %0100 | $4 |
| 5 | %0101 | $5 |
| 6 | %0110 | $6 |
| 7 | %0111 | $7 |
| 8 | %1000 | $8 |
| 9 | %1001 | $9 |
| 10 | %1010 | $A |
| 11 | %1011 | $B |
| 12 | %1100 | $C |
| 13 | %1101 | $D |
| 14 | %1110 | $E |
| 15 | %1111 | $F |

---

## __/Byte & % $ conversion

A Byte consists of 8 bits and is therefor double the width of a nybble. People also refer to the nybble containing LSB as low nybble, to the MSB nybble as high nybble.

The MSB represents the value 128 in decimal. (or 80 in hex system) Largest value of a byte is 255 in dec / $FF in hex / %11111111 in binary.

Buildup of a byte looks like this:

| | MSB | | | | | | LSB |
|---|---|---|---|---|---|---|---|
| bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| dec value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| hex value | $80 | $40 | $20 | $10 | $8 | $4 | $2 | $1 |

The trick with a byte value is, that it can be treated like two nybbles for easy binary to hex (and vice versa) conversion.

%10110110

splits up to

%1011 %0110

furthermore

%1011 = $B

and

%0110 = $6

the outcome: %1011 0110 = $B $6 = $B6

the other way around:

$E4

splits up to

$E = %1110

and

$4 = %0100

outcome: $E4 = %11100100

---

Adding bytes as nybbles is also very easy:

```
  $10  %0001 0000
+ $40  %0100 0000
-----------------
  $50  %0101 0000


  $20  %0010 0000
+ $40  %0100 0000
-----------------
  $60  %0110 0000
```

Though it's somewhat harder when adding to higher values, in that case it's easier to see the byte as "whole" value, not two nybbles.

```
  $7F  %0111 1111
+ $01  %0000 0001
-----------------
  $80  %1000 0000
```

$7F - Low nybble is already at highest value ($F). If this was decimal system, the highest value would be 9. So if you add 79+1, result is 80. In hex just the same, we increase the high nybble ($7) and set low nybble to $0 again. $7F + $01 = $80.

Mind: If you're lazy, you can also use the calculator that comes with your operating system for hex/bin/dec conversion.

---

## __/Signed Values

A signed byte is referred to as signed as it's MSB indicates (signs) if the value is positive or negative. Since there are only 7 bits left for the actual value, it can range from decimal -128 to +127.

Further, the bits are inverted (flipped) in case the represented value is negative. Since %0000 0000 represents 0, a signed value of %1111 1111 represents -1. Also an offset of $01 has to be added after inverting the bits.

Conversion example from positive to negative:

$3E = %0011 1110 (=62)

```
inverted = %1100 0001
+ offset   %0000 0001
----------------------
result     %1100 0010 = $C2 (= -62)
```

And in case you really don't understand the need for the offset, here is the proof:

Converting $01 postive to negative:

$01 = %0000 0001

```
inverted = %1111 1110
+ offset   %0000 0001
----------------------
result     %1111 1111 = $FF (= -1)
```

Therefor signed values $01 to $7F represent 01 to 127,
$FF to $80 represent -01 to -128

Positive Range:

$01=$+01
$02=$+02
$03=$+03
.
.
.
$7D=+$7D
$7E=+$7E
$7F=+$7F

Negative Range:

$FF=$-01
$FE=$-02
$FD=$-03
.
.
.
$82=$-7E
$81=$-7F
$80=$-80

_____/

 _____
/     \

## Chapter 1 - Instrument editor - ADSR - Waveforms & table - Arpeggios

The C64 soundchip (6581/8580) is a little encapsulated synthesizer which is able to create a wide array of gritty and even somewhat organic sounds.

Similar to a church organ, the soundchip also has 27 registers where we can set values to influence, -synthesize- sound. One register contains one byte of data.

Overall the chip has three oscillators, or for the musician: three voices.

Soundgeneration on the sidchip is defined by following factors:

---

Pitch - Frequency. Every Sid voice has two registers for frequency, allowing a 16 bit frequency value. This part/conversion will not be described in deep, as goattracker converts entered notes etc. to frequency values and writes the "pitch" registers by itself.

---

Volume - Amplitude. In synthesizers and also with the sid,
the concept of ADSR curves is used.

## __/ADSR

stands for:

Attack time   -  The time until reaching full volume before decaying
Decay time    -  Time until reaching sustain level
Sustain level -  Sound sustains (Sustain is a level, no time value)
Release time  -  And finally releases, fades out

Or in other words:

Attack  - You press a key on the piano, the hammer hits the string, the
          string resonates with it's peak value
Decay  -  The Peak value decays
Sustain - The string sustains at sustain level (Key still pressed)
Release - Release of the piano key and fade out / muting of the string
          according to foot pedals

Each of these 4 factors is represented by a nybble value which are set in instrument editor. $0 = short, $F = long; in case of sustain $F = loud.

Mind: $F on attack is 8 seconds, $F on decay/release is 24 seconds.

As sound passes -through- this ADSR circuit, it is known as gate.

Bit 0 of the waveform register controls this gate. When it is set to 1 (=gate on), the AttackDecay cycle is initiated, after decay time has ended the sound is held at sustain level. When waveform register bit 0 is set to 0, the release cycle starts.

Diagram for an instrument with fast attack, fast decay and long release:

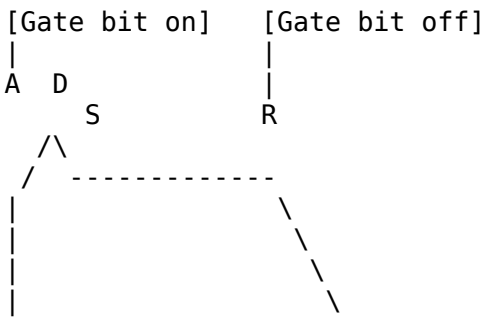
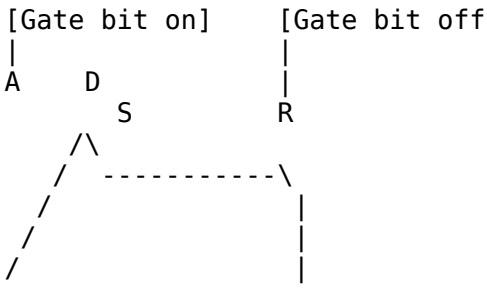



Diagram for an instrument with slow attack, fast decay and short release:




---

## __/Waveform - Wavetable

In goattracker instrument editor, the $ number besides "Wavetable Pos" points to a position in the wavetable, which contains the waveform values which are written frame by frame into the sid registers.

One frame = one screen update = one value read from table and written to sid registers. When people speak of 2x or 4x tunes, they mean that the music routine is called several times per screen update, and thus allows more waveforms and parameters to be written, which results in more abilities to create sound.

The "WAVE TBL" contains two columns, the left value is the waveform register value, the right column is relative/absolute note offset which we will discuss later.

Hint: Use -Tab- to jump between
pattern/orderlist/instrument/wavetable/songname
For direct jumping between instrument editor and wavetable, use F7.

So bit 0 of the waveform register controls the ADSR gate,
what about the other 7 bits?

| | MSB | | | | | | LSB |
|---|---|---|---|---|---|---|---|
| bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| hex value | $80 | $40 | $20 | $10 | $8 | $4 | $2 | $1 |
| dec value | noise | square | saw | tri | test | ring | sync | ADSR |

Bit 7 selects the "noise" waveform, which is very useful for creating hihat and snaredrum sounds. It's no "fixed" noise as for example white noise, It's pitch dependent pseudo random noise defined by a note value in a pattern or by offset/fixed pitch.

Bit 6 "square" or "pulse" waveform, which can be seen as __---__---__---
Mind: you must set pulsetable/pulsewidth,
else you won't hear the pulse waveform (see Chapter 2).

Bit 5 "saw" waveform, which looks like a saw blade /|/|/|.

Bit 4 "triangle" waveform, looks like /\/\/. It has a very soft sound which can be used for flute like sounds.

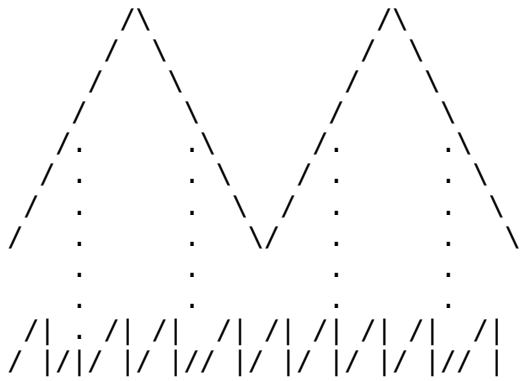
Bit 3 "test" or "reset" bit. C64 oscilltors lock up when selecting waveforms together with noise. This bit is used to unlock it again. You will never need this bit, as goattracker cares for that by itself with it's hardrestart routines.

Bit 2 "ringmodulation" - When this bit is set to 1, the oscillator (it must be triangle waveform!) is ringmodulated with the frequency of the previous oscillator, no matter what waveform the previous osc. is set to. Previous oscillator in goattracker terms means, the osc./pattern left to the modulated voice (the three patterns in the editor represent the three sid voices). In case ringmod bit on voice 1 is selected, the previous voice would be voice 3.
The effect is better heard than described. It sounds metallic and can be used to create bell like sounds.

Bit 1 "sync" - When this bit is set to 1,  the fundamental frequency of the
      oscillator is hardsynced with the frequency of the previous
      oscillator. Technically, the waveform  the oscillator plays is
      restarted when the previous oscillator amplitude passes 50%.

Diagram for sync:

Oscillator 1 playing a triangle wave syncs Osc 2 that plays a saw wave,
with higher pitch than Osc 1:



Bit 0 "ADSR"/ gate

From the stuff we know so far, we will create our first sound:

Open goattracker and jump to Instrument editor [F7].

Edit [Attack/Decay 55]
and  [Sustain/Release 55]

Edit [Wavetable Pos 01]

Jump to WAVE TBL [F7].

Edit
[01:21 00] (Sets saw waveform and gatebit)
[02:20 00] (Saw waveform still set, gatebit off)
[03:FF 00] (FF is a jump command for wavetable programming.
            FF 00 means - stop wavetable execution)

Now hit [space] to hear a test note of that instrument.
(you can use * and / for octave switching)

Since gatebit is on for just one frame, we cannot really hear the attack.

So we change wavetable position 02 to:
[02:FF 01] (jump to position 01)

And hit [space]

When we jump up to instrument editor now, we can change Attack/Decay and
Sustain values and hit [space] to trigger the note once more to hear the
change. Stop sound by pressing [F4].

We can also create complete useless sounds:

Edit
[01:11 00]
[02:21 00]
[03:51 00]
[04:FF 01]

Hint: you can use [insert] and [delete] to move wavetable values up/down.
      If you move onto the 01 position and press [insert] you will see
      goattracker even changes the "Wavetable Pos" according to where you
      move the initial value.

---

__/Arpeggios and fixed pitch sounds
---

As mentioned earlier, the right column in WAVE TBL is used for pitch
offset. This can eighter be relative (=added/substracted from the note
value) or absolute (fixed, no matter what the note pitch is) values.

$00 - $5F positive relative notes
$60 - $7F negative relative notes
$80 - keep frequency unchanged
$81 - $DF - absolute notes C#0 - B-7

Arpeggios - Since the sidchip has a somewhat limited polyphony, the concept
            of arpeggios (single notes that are played in fast order) is
            often used.

If we want to build the guitar typical powerchord by arpeggio, we just need
to count the frets from the initial note to the second (the interval in
half note steps). the third tone is an octave.

In wavetable that would look like:

[01:21 00]
[02:21 07] (+7 halftonesteps =a quint)
[03:21 0C] (+12 halftonesteps =an octave)
[04:FF 01]

Jump to Pattern editor with [F5] and hit space to change to [JAM MODE]
(see lower left of screen) to play the arpeggio with different notes.

Hint: Marking/Painting the "black" piano keys on your QWERTY keyboard makes
      composition a lot easier. There are also two layouts to choose from:
      the fasttracker and the DMC layout. This tutorial is based on the
      (default) fasttracker one.

Hint2: Space (as you've already seen) switches JAM/EDIT mode, so you can
       find out some melody without fuxx0ring the pattern up.

Hint3: If you use a Mac, you can also connect a MIDI keyboard and play
       notes with that :)

If you want to create your own arpeggio scale, just take the base note and
count the keys.

Means: base note key for example [C] (E-3)
       you want second note key [N] (A-3)
       count the keys going up the scale from [C] to [N], not including [C]
       (base note) but [N].
       that would be [V] [G] [B] [H] [N] = $05 relative offset

---

__/Fixed pitch sounds aka drums
---

Values from $81 (C#0) to $DF (B-7) in the right WAVE TBL column play back the sound at the given value, no matter what note value is entered in pattern editor. So to speak: you can trigger these with any note value, it will always be played back at fixed pitch.

Quick example for some oldschool drumsound:

Edit instrument:
[Attack/Decay 32]
[Sustain/Release F9]
[Wavetable Pos 01]

Edit WAVE TBL
[01:51 90] (We combine square and triangle ($40 + $10 = $50), set gate bit
           ($50 +$01=$51) play note with fixed pitch $90)
[02:21 A0] (Now play saw, gate still set, fixed pitch $A0)
[03:80 DF] (Play noise wave, set gatebit to 0 to let the drumsound fade
           out, fixed pitch $DF)
[04:FF 00] (Stop wavetable execution, sound fades out still)

Now jump to pattern editor and enjoy drumsound always in same pitch.

Hint: see the ravenspiral .pdf that comes with goattracker for a nifty note
      value table.

---

__/6581 vs. 8580
---

By design it -should- be possible to combine square, saw and triangle by a logical AND (setting their bits together to 1).
Noise waveform cannot be combined.

Due to bugs (features) in the 6581, this is not completely possible with all waveforms. Combining square and triangle ($50) gives a good audible result, whilst triangle and saw ($30) are nearly inaudible.
Though interesting sounds can be achieved by adding a wide pulse wave.

On 8580 all three waveforms can be combined to give clear audible results.

Mind: Another difference between 6581 and 8580 is the filter. The 6581 is
      out of specs and every chip's filter curve seems different.
      -But- the 6581 filter can be used for really dirty filtering and
      distortion and has a generally more lively sound. The 8580 filter is
      linear to specs and just "does what it should do".
      Decide for yourself which is more important to you.

Trivia: Oldschool musician Ben Daglish didn't use any filters in his songs,
        as he knew every chip sounded different and he did not want that
        his soundtracks would sound completely different on other machines.

_____________________________________________________________________________/

 _____________________________________________________________________________
/                                                                             \

  Chapter 2 - Pulse Waveform - Automation/Modulation

The pulse waveform of the sidchip is somewhat special, as it allows us to
set the relation of the high/low cycle (this is also sometimes called PWM
- pulse width modulation).

Mind: The chip has actually two registers for setting pulsewidth for each
      voice, though one of these two registers uses only a nybble value,
      which gives us a total range from dec 0-4095, or more familiar:
      $000 - $FFF.

A symmetrical square (=pulse) wave would look like this:

```
high  ----    ----    ----    ----
low       ----    ----    ----    ----
```

Now if we change the pulse -width-, it would look like this for example:

```
high  ------  ------  ------  ------
low         --      --      --      --
```

Soundwise this changes the harmonics of the wave, so the sound is missing
certain harmonics whilst it emphasizes others, depending on high/low
relation. It's better heard than described, as it's one of the sid chip's
well known distinct sounds.

So lets create an instrument using PWM.

Edit ADSR values in instrument editor to whatever you like, and set
[Wavetable Pos 01]
[Pulsetable Pos 01] (just like with the wavetable,
                     this points to the PULSETBL position)

Edit WAVE TBL
[01:41 00] (Sets pulse wave and gate on)
[02:FF 00]

As mentioned earlier, we cannot hear anything as we have not set the pulse
high/low relation. So go right to PULSETBL (the pulsetable is executed just
like the wavetable, frame by frame)

Edit
[01:98 00]
[02:FF 00]

and hit [space]. you now hear a symmetric pulse waveform.

01:98 00 means: A value in pulsetable where the most left nybble (in our
case $9) is in range from 8 to F indicates that the
following three nybbles are used to SET the pulsewidth.
So we could also type [01:F8 00],
it would have the same effect.

The second nybble ($8) is the high nybble (MSB) for setting
the pulsewidth, so it influences the pulsewidth the most.
The most right two nybbles are the low byte of the
pulsewidth which allow for finetuning.

edit PULSETBL
[01:9X 00] - try different values for X, hit space to retrigger the sound.

You might have discovered that the sound thins out, the closer you get to $000 and $FFF. That is because a pulse with width $000 or $FFF is so wide that it's constant high or low, thus it does not oscillate and it cannot be heard.

$000 and $FFF are exact opposites, just inverted.

pulsewidth $000
high
low   ----------------------------

pulsewidth $FFF
high  ----------------------------
low

Thats why our initial value $800 gives us a symmetrical square wave, as it's the middle between $000 and $FFF. Going towards $000 from $800 sounds the same as going towards $FFF from $800, though it's not the same seen from the physical side.

---

__/Automation/Modulation - Pulsetable
---

Go to PULSETBL and edit

[01:98 00] (set pulsewidth to $800)
[02:6C 13] (increase pulsewidth with an amount of $13 for $6C frames)
[03:FF 00]

Hit space to hear a pulsewidth fade going from $800 towards $FFF and stop.

The concept of this automation is somewhat odd to use for a musician.

Imagine, the PULSETBL automation is just a table with values, and goattracker (or later the playroutine) adds these values to the pulsewidth register, or substracts them.

How?

[02:6C 13] Values in the pulsetable left side that range from $01 to $7F are used for pulsewidth modulation (=automation).
In our case, $6C $13 means, ADD a value of $13 to the pulsewidth register for the duration of $6C frames. The right byte ($13) is a signed value!

If we change $6C to $1F [02:1F 13] we can hear that the pulse only slightly changes, as $13 is only added for a duration of $1F frames.

If we change $13 to $01 [02:6C 01], the pulsewidth will fade slower, as only a value of $01 is added to the register.

Now we change step 02 to:

[02:7F 20] and hit [space]

You will notice, the sound passes $FFF (the "constant high" pulsewidth where it cannot be heard) and wraps around .

This is because -registers wrap around-; when the value exceeds $FFF and the Pulsetable execution adds another $20, it will start over from $000 and increase again from there until all $7F frames are executed, then it stops again somewhere near $800.

Remind: $7F is the maximum duration for modulation, as a value of $8x and higher is reserved for setting the pulsewidth.

Pulsewidth wrapping graph:

```
$800 __--__--__--__-- (symmetric)
$900
$A00
$B00
$C00 -_---_---_---_-- (3 high/1 low)
$D00
$E00
$FFF ---------------- (silent)
$000 ________________ (silent)
$100
$200
$300
$400 _-___-___-___-__ (1 high/3 low)
$500
$600
$700
$800 __--__--__--__-- (symmetric)
```

Try: The jump command [FF] in Pulsetable. If you change [03:FF 02] the sound will wrap forever. You can also set the width to different values directly in series using the $8x - $Fx left nybble. And you can even execute a fast fade, then a slow fade (several automation commands in series). In that case the starting pulsewidth for a following commands is always the one where the previous command stopped changing the register.

And: The right byte (the value that will be added/substracted) is, as told
     before, a signed value. So if you want to stop at a certain pulsewidth
     and go back reverse, you need to add a negative value (signed) again.
     You should really consider reading -chapter 0 - signed values- if you
     don't know what this means.

Last words for this chapter: Of course the pulsewidth register will also
                             wrap around in opposite direction (like every
                             register). So substracting from $000 will
                             start over from $FFF.

_____________________________________________________________________________/

 _____________________________________________________________________________
/                                                                             \

  Chapter 3 - Usage of Filter - Filtertable

__/Usage of Filter
------------------

Some people call the sid chip a substractive synthesizer. It is
substractive by the means that you can filter/cut off a certain frequency
range of the sound output.

As there are different types of filters, they are differed by the frequency
range they work in. Sid has three different filters, which can be combined
in any way.

Lowpass - It lets low frequencies pass
Bandpass - Lets mid range frequencies pass
Highpass - Guess what that one does?

Also there is ability to set the cutoff frequency register, which defines
at which frequency the selected filter -cuts off-.

So if you select lowpass and set $cutoffvalue, it will let all frequencies
pass that are below $cutoffvalue; so the sound will be somewhat dull.

Bandpass lets frequencies pass that are around $cutoffvalue, so low and
high stuff will not be heard.

Highpass $cutoffvalue will let pass all frequencies above $cutoffvalue, so
no low or midrange sounds can be heard.

Hint: Bandpass is also known as MIDpass.

Furthermore, we have the ability to set a resonance value, which means that
the frequencies in $cutoffvalue range will be emphasized.

_____________________________________________________________________________

## __/Filtertable

So let's create a sustaining instrument with the knowledge we have gained:

Then edit in instrument editor:

[Filtertable Pos 01]

Jump to FILT TBL.

```
edit
[01:90 F7] (left nybble $9x - set lowpass, right unused, $Fx set resonance,
            $x7 set channel/=oscillator)
[02:00 10] (Set cutoff to $10)
[03:FF 00] (end of line)
```

press [space] to hear a muted version of the sound you created.

More deep explanation of the byte values:

Bytes that are written in the left column define either filtertype or !-modulation steps-! (we know them from pulsetable already).

!-Refresh your know-!
When the left byte is in range from $01 to $7F a modulation step will be executed, $01 to $7F defines the number of frames for modulation and the right byte defines the speed/amount (signed value again).

Mind: When you try the modulation examples from [chapter 2 - pulse waveform] here, the value will wrap around as it always does, but the filter sound will not. So finally understanding signed values should be essential for letting filter cutoff going up and down again too.

---

In case the left byte in a filtertable row is above $7F ($80 and up), the whole two bytes in this table step are used for defining the following:

Filter type (left byte)
As we only have 4 values (No filter/LOWpass/MIDpass/HIpass), only the left nybble bits are used ($8 - $F).

This leaves us following bits to switch the filter type:

```
     H  M  L
     I  I  O
     :  D  W
     :  :  :
% 1  0  0  0   $8 - All off, no filter (MSB=always on, since above $7F)
% 1  0  0  1   $9 - LowPass - LP
% 1  0  1  0   $A - MidPass - often abbreviated BP since Mid=Band
% 1  0  1  1   $B - BP & LP
% 1  1  0  0   $C - HighPass - HP
% 1  1  0  1   $D - HP & LP
% 1  1  1  0   $E - HP & BP
% 1  1  1  1   $F - HP & BP & LP
```

Right nybble value has no effect, $80 or $81, even $8F - does the same.

---

Right byte left nybble defines resonance ($0x - $Fx), rightmost nybble
lowest three bits select which chan/voices are to be filtered ($x0 -$x7).

```
    [ voice ]
     3  2  1
     :  :  :
     :  :  :
     :  :  :
% 0  0  0  0   $0 - Filter not active on channels.
% 0  0  0  1   $1 - Voice one filtered.
% 0  0  1  0   $2 – Voice two filtered.
% 0  0  1  1   $3 - 1 & 2 filtered.
% 0  1  0  0   $4 - 3 filtered.
% 0  1  0  1   $5 - 1 & 3 filtered.
% 0  1  1  0   $6 - 2 & 3 filtered.
% 0  1  1  1   $7 - all voices filtered.
```

Now we can totally figure out our filter table example step 01:

```
[01 90 F7] (left nybble $9x - set LP, right nybble unused,
           $Fx set resonance, $x7 set channel/=oscillator)
```

---

In case the left nybble of the step is $00,
like in step 02 from the example:

```
[02:00 10] (Set cutoff to $10) - $00 tells the editor to set filter cutoff
                                 to the right byte value ($10).
```

---

A simple Filtersweep:

```
[01:90 F7] ($90 - set lowpass, $F set resonance, $7 set channels)
[02:00 80] (Set Cutoff to $80 = middle of filter range)
[03:1F FF] (For $1F frames, substract $01 from the Cutoff register)
[04:1F 01] (For $1F frames, add $01 to the Cutoff register)
[05:FF 00]
```

[very] Mind: In the above examples, filter is always selected for all three
channels. That was done deliberate, because the sound you edit
in instrument editor is always played back on the cannel you
have last edited in pattern editor.
Means: when you entered notes on pattern/channel 1 at last and
jump to instrument editor, you will hear the [space] preview
note also played that channel.
Verdict: When you edit a filter and can't hear it, the channel
you are on in pattern editor and the one you set with the
filter voice bits are not the same.

_____/

## Chapter 4 - Vibratos - Speedtable

On the right column of instrument editor we have "Vibrato Param" and "Vibrato Delay".

[Vibrato Param] points to a position in the [SPEEDTBL].

[Vibrato Delay] defines how many frames the sound will play before vibrato.

But what is a vibrato?

It's a small (or big if you set it to that) up and down vibrating change in pitch. It gives the sound more live - makes it sound less static. Violin and guitar players often play vibratos on their instruments by moving the string a little up an down on the fretboard.

So lets edit an instrument  and do the following to add vibrato:

edit [Vibrato Param 01] (As said, points to the speed table)
edit [Vibrato Delay 01] (Adds the vibrato instantly with no delay.
00 here turns vibrato off!)

jump to speedtable
edit [01:0E 0F] and hit [space] to hear a wide (more useless) vibrato.

edit [01:04 05] for a more subtle vibrato.

Speedtable is somewhat different than the other tables. When we use it for vibrato (other uses will be discussed in Chapter 5) the left byte ($04 in the second example) indicates how many frames pitch will be changed before going into the other direction. The right byte ($05 in second example) tells the value that is added/substracted from pitch value.

So the smaller the left value is, the faster the vibrato will be. The bigger the right value is, the wider the pitch change will be. Due to the concept of counting frames before changing pitch into the other direction, a change of the left value also changes pitch.

Maximum value for left byte is $F7 (127 frames before direction change), right byte can go up to $FF. There are no signed values used of course.

Diagram of [01:0E 0F]

Diagram of [01:04 05]

Mind: The speed table is the only table that doesnt work "like a program"
      it does not contain commands that are executed in series. Every linie
      in it is a single entry with no follow up. Therefore we also don't
      need the $FF command here.

_____________________________________________________________________________/

 _____________________________________________________________________________
/                                                                             \
  Chapter 5 - Pattern editing, Pattern commands, Orderlist

__/Pattern editing
------------------

Now that we know how to make an instrument, we can start making a tune.

Mind: +/- Changes - Increases/Decreases instrument number selection.

Up in goattracker window we see:

```
CHN 1 PAT.00  CHN 2 PATT.01  CHN 3 PATT.02
|     |
|     Current Pattern
|     that is loaded
|     and shown below
|
Sid Voice/
Channel

00 ... 00000  00 ... 00000  00 ... 00000
01 ... 00000  01 ... 00000  01 ... 00000
02 ... 00000  02 ... 00000  02 ... 00000
```

So when we jump to pattern editor and enter a note value by keyboard
(for example [N] with fasttracker layout, mind JAM/EDIT mode),
line 00 will change to:

```
00 A-2 01000
   |/| |/\ |
   | | |  \|
   | | |   Pattern Command (Nothing yet)
   | | Instrument number
   | Octave
   Note value
```

You can use cursor keys to scroll up and down and note keys like on piano
to enter note values.

Further, INS and DEL can be used to move notes up/down from the position
where you are editing. ENTER inserts a keyoff, which can be used to cut off
notes are played by an instrument with gatebit never off. With shift+ENTER
a keyon is inserted, so you can even "reincarnate" the wave after a keyoff.

Backspace deletes the note at cursorposition/inserts a rest.

Page up/down lets you move 8 steps up or down at once.

Shift+) or ( changes the pattern on the channel you are currently editing.

Hit INS/DEL on the bottom line of the pattern defines pattern length.

There are much more editing keys and shortcuts you can use, but this would be too much to explain in every tiny detail, and also, every musician tends to have his own way of editing a tune.

Hit [F12] for help screen and try for yourself what you need for editing.

---

## __/Pattern Commands

The last three values in a pattern column are used for pattern commands.

The abbreviations and functions are:

0XY - do nothing

1XY - Portamento (Pitch glide) upwards. XY points to a position in speed table. In this case, the value in speed table is taken as 16 bit value, so maximum glide speed can be $FF FF.

2XY - Portamento downwards. XY points to 16 bit value in speedtable again.

3XY - Toneportamento. Glides pitch from one note to another, XY points to speedtable 16 bit glidespeed. If XY is $00, the notes are played tied together.

Mind: Portamento will only be executed as long as you write the 1/2/3 XY command in command column.

4XY - Adds Vibrato to the voice, XY points to speedtable. same use as with instrument vibrato.

5XY - Set attack/decay register directly to X/Y

6XY - Set sustain/release register directly to X/Y

7XY - Set waveform register directly to value XY. If Wavetable is actively changing waveform at the same time, command will not work.

8XY - Start executing wavetable at position XY. $00 Stops execution.

9XY - Start executing pulsetable at position XY. $00 Stops execution.

AXY - Start executing filtertable at position XY. $00 Stops execution.

BXY - Filter Control, X is resonance, Y is channel selection
00 Turns filter off and also stops filtertable execution.

CXY - Set filter cutoff to XY. Does not work when Filtertable is also

changing cutoff at the same time.

D0Y - Set mastervolume $0-F.

EXY - Funktempo. XY points to speedtable. Alternates between the both
speedtable values (tempo) on each step.

FXY - Set tempo. $03-7F sets tempo on all channels/patterns. $83-FF sets
tempo for current pattern (substract $80 for the actual value).
$03 is fastest. Tempo $00 and $01 recall the funktempos set by
EXY command.

---

## __/Orderlist

The "CHN ORDERLIST" upper right in goattracker window is the actual playlist of patterns. Hit [F6] to go there.

By default it shows:

```
1 00 RST00
2 01 RST00
3 02 RST00
```

If we move the cursor onto [00] and hit INS, we can add our edited patterns into the playlist. DEL removes patterns from the list.

This is somewhat tricky: Hitting ENTER on a pattern number in CHN ORDERLIST will load that pattern into pattern editor. Hitting SPACE on a pattern number will light it green, but not load it into pattern editor.

So if you hit

[F1] the song will be played back from the start of the orderlist.

[F2] will play back the green marked patterns,
no matter if they are shown in pattern editor.

[F3] plays back the patterns that are currently
loaded/shown in pattern editor.

Mind: The steps in pattern editor are decimal (00-63), but pattern numbers are hex. RST00 indicates to ReSTart the song at orderlist position 00.

Idea: Write down which pattern does what in your song
(01=bass, 02=lead etc.).

_____/

## Chapter 6 - Some essential shortcuts/keys/odd things

Repeating some of what we already know, and add some more:

[F4] Normally stops playback, but also stops all sound in instrument editor

[SPACE] Toggles JAM/EDIT mode in pattern editor, triggers note in instrument editor

[F10] Loads song in pattern editor/orderlist, but can load instrument in instrument editor too

[F11] Saves song in patt editor/orderlist, saves single instrument in instrument editor

[Enter] On Wave/Pulse/Filtertable Pos XY takes you directly to that entry in the table

[shift+N] Edit name of instrument in instrument editor

[shift+N] Negates signed values in table modulation steps (Wohoo! You can forget chapter 0 now)

[shift+F8] Switch 6581/8580 sid model

## Chapter 7 - What Chapter 0-6 left out

* When compiling a song to a .sid file [F9], be sure that all data in the tables end with an FF command, else table execution overflow will occur and the song cannot be created.

* It's always a good idea to check your song on a real C64 or using a hardsid card, some bits might sound complete different on the real thing (especially filter).

Thanks to: Bordeaux, mrsid, streetuff

If you find errors in here, have recommendations,
Found something interesting for chapter 7:

mailto: admin@zerozillion.net

Tutorial version 19.07.09@18:11