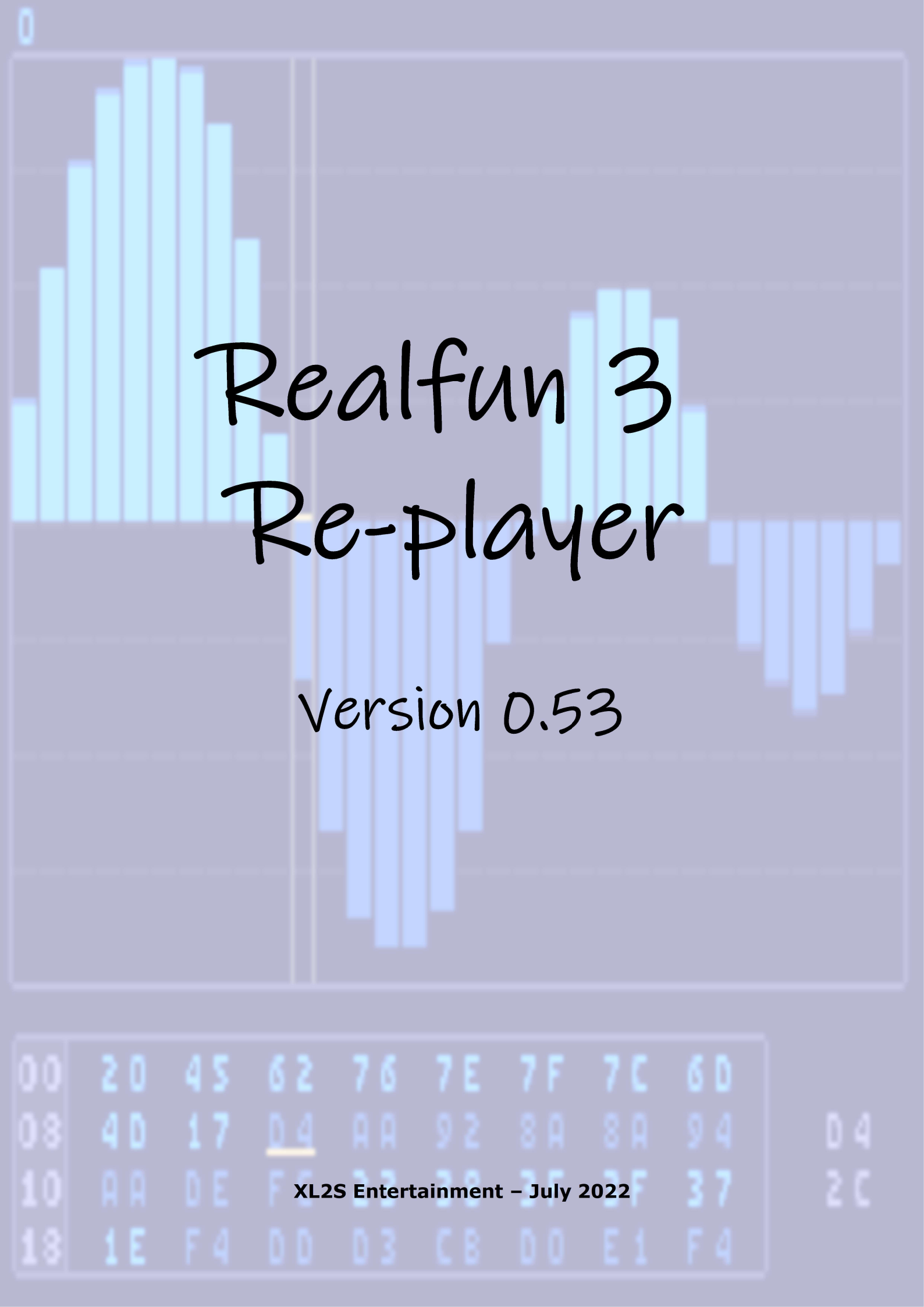

Realfun 3 Re-player

Version 0.53

XL2S Entertainment – July 2022

Getting started

This document will provide an overview of the current features and functionalities of the re-player. Well, enventually, as it is work in progress!

Current version: v0.53

Let's start with mentioning some things:

- There is no SCC detection, this example assumes it is a MegaFlashRomSCC+

- The rom example shows the principles how the re-player can be implemented in an SCC(+) ROM (select MegaFlashRomSCC+ rom type in openMSX), but the whole environment/memory mapping routines and such may need to be revisited and customized anyway. Only a quick and dirty implementation is used now for re-player core testing purposes only. After all the core is what it is all about anyway ;) and how that is implemented in your own program, either running from RAM or ROM, is up to you! Mind the restrictions though ;)

- Realfun 3 re-player song format is non linear and requires random access, for this reason the entire song data (song.sd) needs to be mapped in at once, this will give the following restrictions on song data size (note this is excluding wave or sample data):
  - max. 24 kb using re-player in ROM
  - max. 32 kb using re-player in RAM
- maximum of 512 wave forms per song (i.e. 16 kb of wave form data; song.wd)
- maximum of 16 samples per song, a jump/index table to be manually defined by user, also each sample is to be included by user.

- Since the re-playing format is like a stream (and not really playing the song) in order to make a song loop-able, one has to make sure the same conditions are met in the transition from the last pattern to the loop pattern as the first time going into the loop pattern. Easiest work around/solution for now is to include the loop pattern in the end en make the loop into the second pattern. Can still be tricky for PSG channels with AM and Noise at the moment (and it may be required to do some manually tweaking with for example AM frequency commands to ensure a certain value). Improvements on loop detection in the converter is on the to do list.

- Functionalities other than playing (e.g. fade in/out, signal reading) not implemented yet

- Sound fx / manual triggered samples are not support as of yet

- All coding is made using sjasm assembler and the code will including some of my bad habits like using **add b** instead of **add a,b** among other things thus using another assembler may require some rework :)

- Currently the converter & re-player only supports music files for SCC(+) & PSG (thus no single or dual PSG nor SN support as of yet)

  if one wish to make a regular SCC song use the "sync" command in realfun 3 on the fifth channel to sync all wave updates with channel four. In the re-player comment out the " call scc_wave_stream.run" for channel 5 in the "process_stream_data" routine in file <replay_stream.i> and similarly comment out the " jr nz,.wave_5" in the "update_scc_waves" routine in file <replay_update.i> (as well for channel 5 in <replay_scc_sample.i> if isr samples are used), that should do the trick.

- Some discrepancies in sound may be still be noticeable in some instances, which will be fixed over time hopefully ;)

Example ROM player overview

File overview:

folder	file	description
root	a.bat	assemble rom
	rf3rep.asm	main rom file
code	_variabelen.i	variables for the test rom purposes (not specific to replayer)
	main.i	test rom code
	z_memory.i	some memory sloth select routines (not specific to replayer)
	z_rom_mapper_init.i	rom startup initiation code etc. (not specific to replayer)
replayer	_replay_variables.i	variables for the replayer
	replay.asm	main replayer routines (main address definitions)
	replay_ay_am.i	PSG AM frequency stream processing
	replay_ay_noise.i	PSG noise stream processing
	replay_ay_tone.i	PSG tone stream processing
	replay_scc_sample.i	ISR samples processing
	replay_scc_tone.i	SCC tone stream processing
	replay_scc_wave.i	SCC wave stream processing
	replay_stream.i	Overall channel stream & position table processing
	replay_update.i	Update PSG and SCC registers
songs	(sub folder) andorogy	samples for andorogy song
	(sub folder) new	folder with realfun 3 song files and: _s.bat = batch file to convert the test songs convrf3.exe = converter tool to convert rf3 song to re-player format
	(sub folder) birdfrog	samples for birdfrog song
	a.bat	assemble all re-player format song files
	*.i	re-player format song files
	*.sd	song data files
	*.wd	wave data files

For initiation of the re-player and song follow the steps as shown in beginning on “test” routine in <main.i>.

Note the mapper routines are not fully nor properly implemented, they work for this ROM, but probably need to be customized by the user. Special attention may be required to the “replay.restorerompages” and the memory mapping routines.

To start a song, “replay.start_song” needs to be called with hl pointing to a song info list, which has the following makeup:

byte	start rom page for wave data	(re-player swaps in 2 pages => 16 kb)
word	start address wave data	address to be aligned per 256 bytes!
byte	start rom page for song data	(re-player swaps in 3 pages => 24 kb)
word	start address song data	address to be aligned per 2 bytes!
byte	rom page with sample_jump_table	(if 0 => no isr samples)
word	address of sample_jump_table	address to be aligned per 2 bytes!

Note: if song data starts other than on 4000h be aware that the phase address needs to be adjusted using the [-p] parameter while converting the song (see song converter section).

After the initiation, the song can be played by calling the following two main functions:

replay.play_out	Updates all registers for the sound chips that were prepared and set ready for update (periods, volumes and wave forms etc.)
replay.play_process	Music data processing (set everything ready for next update)

The memory mappings for each routine is as follows:

Music data processing

```
ram / bios
        0000 – 3FFF
rom bank
  1     4000 – 5FFF  song data  (song.sd)        / sample data
  2     6000 – 7FFF  song data  (song.sd)        / sample data
  3     8000 – 9FFF  song data  (song.sd)
  4     A000 – BFFF  re-player (music data processing routines from A000-B7FF) *
ram
        C000 – C4FF  re-player variables (aligned 256 bytes)
```

In case of sample playback, rom bank 1 and 2 will be used to map in the sample data.

Register update

```
ram / bios
        0000 – 3FFF
rom bank
  1     4000 – 5FFF  wave form data  (song.wd) / sample data
  2     6000 – 7FFF  wave form data  (song.wd) / sample data
  3     8000 – 9FFF  *
  4     A000 – BFFF  re-player (register update routines from A000-B7FF,
                     SCC+ from B800-BFDF) *
ram
        C000 – C4FF  re-player variables (aligned 256 bytes)
```

In case of sample playback, rom bank 1 and 2 will be used to map in the sample data.

** In case of regular SCC the re-player should be mapped in bank 3 (8000 – 9FFF) during register update to access the SCC . Note that for that scenario the routines should be assembled with the right phase, i.e. all music processing related routines in phase A000-B7FF and register update related routines in phase 8000 – 97FF, which should be a straight forward split.*

Of course alternative options, for example using re-player and song data in RAM, is possible. It will require some custom modifications here and there but the following principles remain the same:

- re-player variables should be accessible in 0000 – 3FFF or C000 – FFFF range
- song data up to max. 32kb in 4000-BFFF range
- wave data up to max. 16kb in 0000-7FFF or C000-FFFF range, note the 8000-BFFF needs to reserved in case of external SCC or SCC+ is used.

An other range for song data may be feasible (e.g. 0000-7FFF) but require some alterations in the stream "call" and "jump" routines as well as the macro generated by the song converter.

Re-player register usage

The re-player uses only the “default” registers a, bc, de and hl. The alternative and index registers are not used.

Re-player variables

An overview of the variables area is given below. Note that the areas designated as “free” can be used freely by the user.

The variable area can be placed anywhere in RAM (aligned per 256 bytes) as long it is accessible by the re-player at the time of processing or register updates.

	00h	22h	32h	49h	C0h
C000	SCC channel 1	AY channel 1	Control	Reserved	Sample jump table
C100	SCC channel 2	AY channel 2	Free		
C200	SCC channel 3	AY channel 3	Free		
C300	SCC channel 4	AY noise	Free		
C400	SCC channel 5	AY AM frequency	Free		

Song converter

The converter converts the realfun 3 song file (song.r3m) into the re-player format file (song.r3m.i), this output is a text file that needs to be assembled in a assembler (sjasm). When assembled it will give a wave data file (song.wd) and a song data file (song.sd).
In case of samples the converter extracts the required samples and generate the sample files (samplekit_file.sam*).

The converter usage is as follows:

convrf3 song_file [samplekit_file] [arguments[]]

Where the following arguments can be added:

-r = print song interpretation by converter in readable text format
-2 = split patterns in half length (may sometimes result in shorter songs)
-i.. = assigning isr sample (0 to f) as "instrument" (example: -i12ef)
-o.. = assigning isr sample (0 to f) as "on-the-fly" (example: -o45)
-m = file generation for earlier RAM re-player version (no output & phase commands)
-e = extract all samples from the samplekit regardless if used or not
-p.... = change song phase address (example: -p4a00 => phase 4a00h; default: 4000h)

examples:

convrf3 awesome.r3m awesome.r3w -i0123456789abcdef	(all samples as instruments)
convrf3 test0.r3m andorogy.r3w -r -od	(sample "d" as on-the-fly)
convrf3 wildarm.r3m -2	(use halve pattern lengths)
convrf3 doctor.r3m	(all default)

SCC ISR samples

There are three types of isr sample encoding:

default	pre-calculated period data embedded in song data (except in the case of C-4 note, then period data is read from sample data)
instrument	all data of sample and pre-calculated period data embedded in wave and song data
on-the-fly	base note embedded in song data, periods from sample data are re-calculated on the fly (note: calculations are skipped for C-4 base note)

ISR samples as instrument is the fastest method, as it does not require the sample playback overhead. On-the-fly is the slowest in case of pitch shifting as the calculations are quite expensive, but for one or two samples at a time this can be acceptable.
For short samples as instrument or default is therefore recommended, for very long samples, with a lot of period data, on-the-fly may be better, especially in case of a pitch shift (i.e. deviation from C-4 base note). In case of no pitch shift (thus only base C-4) default setting can be used for long samples as well.

After song conversion a sample overview (sample info) is given in the song list file. This overview shows witch samples are used, the sample type and the data source. In case of "**rom**" the sample data needs to be included manually in the rom and a sample_jump_table needs to be prepared by the user to let the re-player know where the samples can be found. This table is 16 x 4 = 64 bytes long (Format: rom page, address low, address high, 00h).

Note that the samples extracted by the converter are excluding the TT sample header (first 6 bytes) and can be included in the rom directly. If TT samples are used these first 6 bytes need to be skipped manually (i.e. use incbin TTsample.sam,6 see Realfun 3 manual).

Re-player song format

(output from converter: song.r3m.i)

```
; Realfun 3 replay format                    ; header
; song title: Song title                     ; song name

  ifndef cad                                 ; some fancy macro for call & jump encoding
  macro cad 1
  exitmacro low ( high ( @1-4000h)*2+1),low @1
  endm
  endif

  module song                                ; module

;   word position                            ; commented out by default,
                                             ; used for RAM re-player (-m argument)

;-----------------------------------------------------------------------------------------------
; sample info
;
;  #  used type        data                 ; included only if ISR samples are used in the song
; --------------------------                 ; indicates witch samples are used and assigned type
; 00  no   ------------  ----
; 01  yes  instrument  wav                   ; wav = sample data included in wave data
  ...
; 0E  yes  on-the-fly   rom *                ; rom * = sample data to be included in rom by user
; 0F  yes  default      rom *



  output test0.wd                            ; start output of wave data part

;-----------------------------------------------------------------------------------------------
sccwaves


   wave data included here                   ; 32 bytes per wave form (up to max 512 wave forms)



  output staff.sd                            ; start output of song data part

  phase 04000h                               ; set song in correct address range (for calls / jumps)

;-----------------------------------------------------------------------------------------------
position


  position table included here               ; to be word aligned !


  first word is a pattern length in ticks,   if pattern length is 0 then follow by loop address
  otherwise followed by channel stream pointers where the following ID is used:

        c       = scc tone stream pattern list (cl) or tone stream part (cp)
        w       = scc wave stream pattern list (wl)
        a       = psg (ay) tone stream pattern list (al) or tone stream part (ap)
        n       = psg (ay) noise stream pattern list (nl) or noise stream part (np)
        m       = psg (ay) AM freq stream pattern list (ml) or AM freq stream part (mp)
```

;---
sccpatlist

all scc pattern tone lists included here (cl)

;---
sccparts

all scc call tone parts included here (cp)

;---
sccwavlist

all scc wave update lists included here (wl)

;---
aypatlist

all psg (AY) pattern tone lists included here (al)
;---
ayparts

all psg (AY) call tone parts included here (ap)

;---
ay_noise_patlist

all psg (AY) pattern noise lists included here (nl)

;---
ay_noise_parts

all psg (AY) noise call parts included here (np)

;---
ay_am_patlist

all psg (AY) pattern AM frequency lists included here (ml)

;---
ay_am_parts

all psg (AY) AM frequency call parts included here (mp)

;---

dephase

endmodule

- e n d o f f i l e -

Re-player song format (byte streams encoding)

>> denotes additional byte fetch from the stream

SCC tone channel stream

```
wait   |     1 - 30    | 000|        - value >= 9  =>     wait 1 – 30 ticks
                                       value      8  =>     extended wait
                                                            >> byte   wait 1 – 256 ticks

set 0  |   0 – 15  |0 | 100|        - set (absolute) volume (0-15)

       |   2 – 10  |1 | 100|        - update ISR sample timer (0-8),   (execute next)
       |      1    |1 | 100|        - switch period re-calculation off,  (execute next)
       |      0    |1 | 100|        - stop sample,                       (execute next)

set 1  |freq high |sm| 110|        - set (absolute) period & volume (& start sample)

                        >>     freq low byte
                        >>     volume byte
                  if sm >>     sample control byte

                  I S R   s a m p l e   c o n t r o l   b y t e

                  | p |  sn | tmr |       - start sample, where:

                                          p    =  period re-calculation on/off (1/0)
                                          sn   =  sample number (0-15)
                                          tmr =  timer (0-7), which is timer 1 to 8

set 2  | ivo | ifq |rp  | 010|        - set (relative) period & volume

                  escape code 10       = not used
                  escape code 2        = repeat last set 2 command multiple times
                                           >> byte   repeat  1 – 255 times
                  rp      0 – no repeat
                          1 – repeat this command once

                  ifq     00 – no update
                          10 – set value & add          (after execution ifq = 11, for repeat)
                               >> byte   signed value
                          01 – add negated value
                          11 – add value

                  ivo     00 – no update
                          10 – add signed value         (not used in combination with repeat)
                               >> byte   signed value
                          01 – increase volume
                          11 – decrease volume

call   | high pointer  |  1 |        - call stream part, or jump to part when in call

                   >>   byte   low pointer

ret    | 0 0 0 0 0 0 0 0 |          - return to stream address before call
```

SCC wave channel stream

inc	|00 |xxx| bbb|	increase wave no. xxx (0-7) times, followed by bbb (0-7) number of blanks
dec	|01 |xxx| bbb|	decrease wave no. xxx (0-7) times, followed by bbb (0-7) number of blanks
wait	|100 | b bbbb|	number of blanks (1-32)
rep.	|101 | n nnnn|	repeat previous inc/dec wave command (1-32) times
set	|11|w|r| bbbb|	set wave number r = 1 => use previous set wave number (i.e. re-trig instrument) r = 0 set new wave, where: >> byte wave number (0-255) w = wave number high (i.e 0-255 or 256-511) update followed by bbbb (0-15) number of blanks
stop	| 00 000 000 |	done for this pattern

(I know, for some mysterious reason I used left aligned bits here... :)

PSG (AY) tone channel stream

wait | 1 - 30 | 000| - value >= 9 => wait 1 – 30 ticks
value 8 => extended wait
>> byte wait 1 – 256 ticks

set 0 | am| 0 – 15 | 100| - set (absolute) volume (0-15) & am bit

set 1 | freq high |0 | 110| - set (absolute) period & volume

>> freq low byte
>> volume byte (+ am bit)

set 3 |-- |ns |tn | 1| 110| - set tone and noise on/off bits (*execute next*)

set 2 | ivo | ifq | rp | 010| - set (relative) period & volume

escape code 10 = not used
escape code 2 = repeat last set 2 command multiple times
>> byte repeat 1 – 255 times

rp 0 – no repeat
1 – repeat this command once

ifq 00 – no update
10 – set value & add (*after execution ifq = 11, for repeat*)
>> byte signed value
01 – add negated value
11 – add value

ivo 00 – no update
10 – add signed value (*not used in combination with repeat*)
>> byte signed value
01 – increase volume
11 – decrease volume

call | high pointer | 1 | - call stream part, or jump to part when in call

>> byte low pointer

ret | 0 0 0 0 0 0 0 0 | - return to stream address before call

PSG (AY) noise channel stream

wait | 1 - 62 | 00| - value >= 5 => wait 1 – 62 ticks
value 4 => extended wait
>> byte wait 1 – 256 ticks

set 0 | 0 – 31 |0 | 10| - set noise period

set 1 |- | 0 – 15 |1 | 10| - set signal (0-15) (*execute next*)

call | high pointer | 1 | - call stream part, or jump to part when in call

>> byte low pointer

ret | 0 0 0 0 0 0 0 0 | - return to stream address before call

PSG (AY) AM frequency channel stream

wait | 1 - 62 | 00| - value >= 5 => wait 1 – 62 ticks
value 4 => extended wait
>> byte wait 1 – 256 ticks

set 1 |shape|us |uf |110| - set (absolute) shape, frequency

if uf >> period low byte
>> period high byte

if us set shape (0-7 = shape 8-F)

set 2 | shp| ifq |rp |010| - set (relative) frequency and set/re-trig shape

escape code 10 = not used
escape code 2 = repeat last set 2 command multiple times
>> byte repeat 1 – 255 times

rp 0 – no repeat
1 – repeat this command once

ifq 00 – no update
10 – set value & add (*after execution ifq = 11, for repeat*)
>> byte signed value
01 – add negated value
11 – add value

shp 00 – no update
01 – re-trig shape
11 – new shape (*not used in combination with repeat*)
>> byte new shape

(10 - set shape with check to be included?)

call | high pointer | 1 | - call stream part, or jump to part when in call

>> byte low pointer

ret | 0 0 0 0 0 0 0 0 | - return to stream address before call