HOW TO WRITE A 1-BIT MUSIC ROUTINE

Tutorial

UTZ[1]

CONTENTS

1 The Basics 2
- 1.1 Pluse Frequency Modulation . . . 2
- 1.2 Pulse Interleaving . . . 3

2 Loader / Wrapper 3

3 Calculating Note Counters 5

4 Counting in 16-bits 5

5 Improving the Sound 7
- 5.1 Timing Issues . . . 7
- 5.2 Row Transition Noise . . . 8
- 5.3 Discretion Noise . . . 9
- 5.4 I/O Contention . . . 9

6 Drums 10

7 Variable Pulse Width 12

8 More Soundloop Tweaks 13
- 8.1 Accumulating Pin Pulses . . . 13
- 8.2 Skipping Counter Updates . . . 14

9 Achieving Simple PCM with Wavetable Synthesis 15

10 Sound Tricks - Noise, Phasing, SID Sound, Earth Shaker, Duty Modulation 18
- 10.1 Noise . . . 18
- 10.2 Phasing . . . 19
- 10.3 SID Sound . . . 19
- 10.4 Earth Shaker . . . 20
- 10.5 Duty Modulation . . . 20

11 Synthesizing Basic Waveforms: Rectangle, Triangle, Saw 21

ABSTRACT

Hiya folks, in this tutorial I'm going explain how you can write your own multi-channel music routines for 1-bit devices. If you have any questions or suggestions, feel free to post in this thead anytime wink

There are various 1-bit synthesis methods. I'm going to demonstrate only the two most common ones here. For explanation I'll mostly use my own flavour of pseudo code, and parallel to that I'll give real-life Z80 asm examples, as it would be done on a ZX Spectrum computer.

[1] *http://randomflux.info/1bit/viewtopic.php?id=21*

1 THE BASICS

1.1 Pluse Frequency Modulation

Method 1 uses a synthesis procedure called Pulse Frequency Modulation (PFM) at it's heart. Because of the thin, razor-like pulses it produces, PFM is also known as the "pin pulse method". It is used in many engines like Octode, Qchan, Special FX/Fuzz Click, or Huby. The approach allows for mixing of many software channels even on slow hardware, but usually does not reproduce bass frequencies very well.

Ok, let's take a look at how the method works. Assume we have the following variables:

- **counter1** - a counter which holds the frequency value for channel 1. On Spectrum, let's use register B.
- **counter2** - a counter which holds the frequency value for channel 2. On Spectrum, let's use register D.
- **backup1** - a copy of the initial value of counter1. On Spectrum, let's use register C.
- **backup2** - a copy of the initial value of counter2. On Spectrum, let's use register E.
- **state** - the output state of channel 1, can be off (0) or on (1). On Spectrum, we'll use A.
- **timer** - a counter which holds the note length. Let's use HL for that.

So, in order to synthesize our two software channels, we do the following:

```
       di               ; DISABLE INTERRUPTS

soundLoop:
       xor a            ; state := off
       dec b            ; DECREMENT counter1
       jr nz,skip1      ; IF counter1 == 0 THEN
       ld a,0x10        ;   state := on
       ld b,c           ;   counter1 := backup1
skip1:                  ; ENDIF
       out (0xfe),a     ; OUTPUT state1

       xor a            ; state := off
       dec d            ; DECREMENT counter2
       jr nz,skip2      ; IF counter2 == 0 THEN
       ld a,0x10        ;   state := on
       ld d,e           ;   counter2 := backup2
skip2:                  ; ELSE
       out (0xfe),a     ;   OUTPUT state2

       dec hl           ; DECREMENT timer
       ld a,h           ; IF timer != 0 THEN
       or l
       jr nz,soundLoop  ;   GOTO soundLoop

       ei               ; ENABLE INTERRUPTS
       ret              ; EXIT
```

1.2 Pulse Interleaving

Method 2 is called Pulse Interleaving, or XOR method. It is used in engines like Tritone, Savage, Wham! The Music Box, and Phaser1. This method will generate a more classic chiptune sound with full square waves and good bass. The drawback of this approach however is that it is more limiting in terms of the number of channels that can be generated.

Assume we have the same variables as in example 1. However, this time we'll use the H register to keep track of state1, and L to keep track of state2. That means we can't use HL as our timer anymore. Well, luckily we have IX at our disposal as well.

In addition, we need a constant which holds a value that will toggle any output state between off and on. We'll call it ch-toggle. On Spectrum, a value of 10h or 18h will do the trick.

```
       ld h,0            ; state1 := off
       ld l,0            ; state2 := off
       di                ; DISABLE INTERRUPTS

soundLoop:
       dec b             ; DECREMENT counter1
       ld a,h            ; LOAD state1
       jr nz,skip1       ; IF counter1 == 0 THEN
       xor 0x10          ;   state1 := state1 XOR ch-toggle
       ld h, a
       ld b,c            ;   counter1 := backup1
skip1:                   ; ENDIF
       out (0xfe),a      ; OUTPUT state1

       dec d             ; DECREMENT counter2
       ld a,l            ; LOAD state2
       jr nz,skip2       ; IF counter2 == 0 THEN
       xor 0x10          ;   state2 := state2 XOR ch-toggle
       ld l, a
       ld d,e            ;   counter2 := backup2
skip2:                   ; ENDIF
       out (0xfe),a      ; OUTPUT state2

       dec ix            ; DECREMENT timer
       ld a,ixh          ; IF timer != 0 THEN
       or ixl
       jr nz,soundLoop   ;   GOTO soundLoop

       ei                ; ENABLE INTERRUPTS
       ret               ; EXIT
```

2 LOADER / WRAPPER

In previous chapter, I talked about the two different synthesis methods commonly found in 1-bit/beeper engines. Now, in order to transform these synthesis cores into an actualy 1-bit player, you'll need to add some code to load in the desired frequency values.

First, you should think about the layout of your music data. It's common to use a two-part structure. The first part is the song sequence, which is actually a list of

references to the pattern data which follows in part 2. So, an example music data file could look like this:

```
sequence {                              sequence:
   pattern00                                  dw pattern00
   pattern01                                  dw pattern01
   pattern02                                  dw pattern02
   pattern03                                  dw pattern03
   sequence_end                               dw #0000
}

lpattern00 {                            pattern00:
   1st note ch1, 1st note ch2                 db nn,db nn
   2nd note ch1, 2nd note ch2                 db nn,db nn
   3rd note ch1, 3rd note ch2                 db nn,db nn
   ...                                        ...
   pattern_end                                db #ff
}

pattern01 {                             pattern01:
   ...                                        ...
}

...                                     ...
```

Calculating Note Counters Reading in data like this is theoretically quite trivial, but may get a bit confusing if you have to write the loader in assembly language.

For the following example, we'll use the usual counter1, counter2, backup1, backup2, and timer variables from the example in part 1. In addition, we'll need two pointers:

- **seq** - will be our pointer to the song sequence
- **pat** - will be our pointer to the current pattern.

Now, you'll need to do the following:

```
init:
      ld hl,sequence          ;    seq := sequence+0
      push hl

read_sequence:
      pop hl                  ;    pat := (seq)
      ld e,(hl)
      inc hl
      ld d,(hl)
      xor a                   ;    IF pat == sequence_end THEN
      or d
      ret z                   ;    .. EXIT
      inc hl                  ;    INCREMENT seq
      push hl
      push de

read_pattern:
      pop de                  ;    counter1 := (pat)
      ld a,(de)
      ld b,a
      cp 0xff                 ;    IF counter1 == pattern_end THEN
      jr z,read_sequence      ;      GOTO read_sequence
```

```
ld c,b                  ;    backup1 := counter1
inc de                  ;    INCREMENT pat
ld a,(de)               ;    counter2 := (pat)
inc de                  ;    INCREMENT pat
push de
ld h,a                  ;    backup2 := counter2
ld l,h

call soundLoop          ;    CALL soundLoop
jr read_pattern         ;    GOTO read_pattern
```

Note that this doesn't set the timer - I think you can figure out yourself how to do that.

3 CALCULATING NOTE COUNTERS

So you've got your sound loop and your data loader all set up, but now you need to actually fill in some music data. For that, you will need to know how to calculate the note/frequency counter values.

Basically, all you need to know is the magic formula:

```
fn = int(f0 / (a)^n)
```

where

- **fo** is the base note counter value you want to use
- **n** is the distance to the base counter value in halftones
- **fn** is the frequency of the note n halftones away from the base note
- **a** is the twelth root of 2, approx. 1.059463094

Unless you want concert pitch, it doesn't really matter so much which base value (fo) you use, so you might as well use something high, e.g. 254. Now, to calculate the value for the base note plus one halftone, you do:

```
f1 = int(254 / 1.059463094^1) = 239
```

For the base value plus two halftones, it's

```
f2 = int(254 / 1.059463094^2) = 226
```

As you can see, the lower the note, the higher the counter value. This makes sense because we decrement these values in the sound loop. A higher value means it takes longer for the counter to reach zero and reset, therefore the output is activated/toggled less frequently - which of course results in a lower frequency.

4 COUNTING IN 16-BITS

If you have tried the methods in Part 1, you might have noticed that it produces a lot of detuned notes at higher frequencies. This is because 8-bit values are too small to properly represent the common range of musical notes.

So, in order to increase the usable tonal range of your engine, you should use 16-bit values for your frequency counters. However, this poses another problem: As 1-bit DACs are usually hooked up to slow 8-bit CPUs, 16-bit maths are generally rather

slow in execution. So simply decrementing our frequency counters like in Part 1 will most likely be too slow. We therefore need to use a trick to speed up counting.

The trick, in this case, is to add up counters repeatedly and check for carry (ie, see if the result was >FFFFh.) I'll explain how this works for the Pulse Interleaving method. We need the following variables:

- **base1** - the base frequency of channel 1. We'll put this in memory on ZX Spectrum.
- **base2** - the base frequency of channel 2. We'll put this in memory on ZX Spectrum.
- **counter1** - the actual frequency counter of channel 1. We'll use BC on the Spectrum.
- **counter2** - the actual frequency counter of channel 2. We'll use DE on the Spectrum.
- **state1** - output state of channel 1. Let's use IYh.
- **state2** - output state of channel 2. Let's use IYl.
- **timer** - note length counter. We'll use IX.

and our usual ch-toggle constant.

```
       di                ; DISABLE INTERRUPTS
       ld iy,0           ; state1 := 0
 ; state2 := 0
       ld bc,0           ; counter1 := 0
       ld de,0           ; counter2 := 0

soundLoop:
       ld hl,nnnn        ; nnnn = base1
       add hl,bc         ; counter1 := base1 + counter1
       ld b,h
       ld c,l
       jr nc,skip1       ; IF previous operation resulted in carry
       ld a,iyh          ;   state1 := state1 XOR ch_toggle
       xor 0x10
       ld iyh,a
skip1:                   ;   ENDIF
       ld a,iyh          ; OUTPUT state1
       out (#fe),a

       ld hl,nnnn        ; nnnn = base2
       add hl,de         ; counter2 := base2 + counter2
       ex de,hl
       jr nc,skip2       ; IF previous operation resulted in carry
       xor 0x10          ;   state2 := state2 XOR ch_toggle
       ld iyl,a
       ld a,iyl
skip2:                   ; ENDIF
       ld a,iyl          ; OUTPUT state2
       out (#fe),a

       dec ix            ; DECREMENT timer
       ld a,ixh          ; IF timer == 0 then
       or ixl
       jr nz,soundLoop   ; .. GOTO soundLoop
```

```
       ei                ; ENABLE INTERRUPTS
       ret               ; EXIT
```

Of course the above asm code can be optimized further, but that I will leave to you, the programmer.

Beware that in order to calculate the counter values, you will need to adapt the formula from previous chapter. Simply change it to

```
fn = f0 * (a)^n
```

5 IMPROVING THE SOUND

You've followed this tutorial series and have come up with a little 1-bit sound routine of your own design. Only problem - it still sounds crap - notes are detuned, there are clicks and crackles all over the place, and worse of all, you hear a constant high-pitched whistle over the music. So, it's time to address some of the most common sources of unwanted noise in 1-bit sound routines, and how to deal with them.

5.1 Timing Issues

In order to keep the pitch stable, you need to make sure that the timing of your sound routine is as accurate as possible, ie. that each iteration of the sound loop takes exactly the same time to execute.

Let's take a look again at the first example from chapter 1. We can see that that code is not well timed at all:

```
soundLoop:

       xor a                    ;4
       dec b                    ;4
       jr nz,skip1              ;12/7
       ld a,#10                 ;7
       ld b,c                   ;4
skip1:
       out (#fe),a              ;11

       xor a                    ;4
       dec d                    ;4
       jr nz,skip2              ;12/7
       ld a,#10                 ;7
       ld d,e                   ;4
skip2:
       out (#fe),a              ;11

       dec hl                   ;6
       ld a,h \ or l            ;8
       jr nz,soundLoop          ;12
;78/92t
```

Due to the two jumps, there are four different paths through the core (no jump taken, jump 1 taken, jump 2 taken, both jumps taking), and the sound loop length thus varies up to 12 t-states - that's more than 15% of the sound loop length, and therefore clearly unacceptable.

We need tomake sure that the sound loop will always take the same amount of time regardless of the code path taken. One possible solution would be to introduce an additional time-wasting jump:

```
soundLoop:

       xor a                    ;4
       dec b                    ;4
       jr nz,wait1              ;12/7----
       ld a,#10                 ;7
       ld b,c                   ;4
       nop                      ;4-------7+7+4+4=22
skip1:
       out (#fe),a              ;11

       xor a                    ;4
       dec d                    ;4
       jr nz,wait2              ;12/7
       ld a,#10                 ;7
       ld d,e                   ;4
       nop                      ;4
skip2:
       out (#fe),a              ;11

       dec hl                   ;6
       ld a,h \ or l            ;8
       jr nz,soundLoop          ;12
;100/100t

wait1:
       jp skip1                ;12+10=22

wait2:
       jp skip2
```

There are other possibilities, but I'll leave that for another part of this tutorial.

5.2 Row Transition Noise

A common moment for unwanted noise to occur is ironically not during the sound loop, but between notes - the moment when you're reading in new data and updating your counters etc. This is called row transition noise.

Row transition noise is very difficult to avoid. Your focus should therefore be on reducing transition noise rather than trying eliminating it. The key to this is to read in data as fast and efficiently as possible. Not much else can be said about this, except: Make sure you optimize your code. For starters, WikiTI has an excellent article on optimizing Z80 code (http://wikiti.brandonw.net/index.php?title=Z80_Optimization).

Theoretically, there is a way for eliminating transition noise, though in practise very few existing beeper engines use it (Jan Deak's ZX-16 being a notable example). That way is to do parallel computation, ie. read in data while the sound loop is running. Obviously this is not only rather difficult, but also it is usually only feasible on faster machines - on ZX Spectrum, it will most likely slow down your sound loop too much.

Which brings us to another problem...

5.3 Discretion Noise

Discretion noise, also known as parasite tone, commonly takes the form of a high-pitched whistling, whining, or hissing. It inevitably occurs when mixing software channels into a 1-bit output and cannot be avoided. It is usually not a big deal when doing PFM, but can be a major hassle with Pulse Interleaving. The solution is to push the parasite tone's frequency above the audible range. In other words, if you hear discretion noise, your sound loop is too slow. As a rule of thumb, on ZX Spectrum (3,5 MHz) your sound loop should not exceed 250 t-states.

Let's take a look at the asm example from chapter 4 again. At the end of the sound loop, there is a relative jump back to the start (jr nz,soundLoop). A better solution would be to use an absolute jump (jp nz,soundLoop) instead, because an absolute jump always takes 10 t-states, but a relative jump takes 12 if the jump is actually taken, which we assume to be the case here.

Also, leading up to the jump we have:

```
; .. before the jump

       dec ix
       ld a,ixh
       or ixl
       jr nz,soundLoop
```

which takes a whopping 38 t-states. It may be a good idea to replace it with

```
; .. before the jump after changes

       dec ixl
       jr nz,soundLoop
       dec ixh
       jp nz,soundLoop
```

This will take only 20 t-states except when the first jump is not taken. It will introduce a timing shift every 256 sound loop iterations, but this is usually not a major problem, as it happens at a frequency below audible range.

I'll cover some more tricks for speeding up synthesis in one of the following parts.

5.4 I/O Contention

This section addresses a problem that is specific to the ZX Spectrum. You can most likely skip this section if you're targetting another platform.

IO Contention is an issue that occurs on all older Spectrum models up to and including the +2. The implication is that in certain circumstances, writing values to the ULA will introduce an additional delay in the program execution. You don't need to understand the full details of this, but if you are curious you can read all about IO contention here.

What's important to know is that delay caused by IO contention affects our sound loop timing. Which is bad, as I've explained above. For sound cores with only one OUT command the solution is rather trivial: You just need to make sure that the number of t-states your sound loop takes is a multiple of 8. For ideal sound in cores with multiple OUTs however, the timing distance between each OUT command must be a multiple of 8. Naturally this is pretty tricky to achieve (and chances are your core will sound ok without observing this), but keep it in mind as a general guideline.

6 DRUMS

We got ourselves a nicely working 1-bit routine now, but something is missing. Now what could that be? Oh right, we need some drums!

As usual, there are several approaches to realize drum sounds. The by far most common one is the "interrupting click drum" method. The idea is that in order to play drum sounds, you briefly pause playback of the tone channels and squeeze the drums in between the notes. In order for listeners to not realize that tone playback has been interrupted, the drum sounds need to be quite short, typically in the range of a few hundred up to a couple of thousand t-states.

There are countless ways of actually producing the drum sounds - pretty much anything that makes noise goes. I'll only post a very primitive example here to get you started, the rest is entirely up to your creativity wink

We'll need 3 variables:

- **$data_{pointer}$** - a pointer into ROM or another array of arbitrary values. On ZX Spectrum, we'll use HL.
- **timer** - a counter to keep track of the drum length. We'll use B on the Speccy.
- **state** - our good friend, the output state. Let's use the accumulator A.

and a constant which equals the state being on/1, let's call it ch-on.

Now we do the following:

```
drumloop:           ;
    ld a,(hl)       ;  state := (data-pointer) AND ch-on
    and 0x10        ;
    out (#fe),a     ;  OUTPUT state
    inc hl          ;  INCREMENT data-pointer
;                   ;  DECREMENT timer
    djnz drumloop   ;  IF timer != 0 THEN
;                   ;    GOTO drumloop
;                   ;  ELSE
    ret             ;     EXIT
;                   ;  ENDIF
```

This will create a very short noise burst - for better sound, you may want to add some bogus commands for wasting a few cycles in the loop. You would typically trigger this code at some point during reading in data for the next sound loop.

One last thing, you will need to adjust the main soundloop timer. Otherwise you will get an unwanted groove effect every time a drum is played. So you need to count the number of t-states your drum code takes to execute. Divide this number by the amount of t-states your sound loop takes, and subtract the result from the main timer every time you trigger a drum.

Another approach to creating drums is the "PWM sample" method. PWM (pulse-width modulation) samples are a distant relative of the more widely known PCM (WAV) samples. In PCM data, each data value (also known as sample) represents the relative volume at a given time. However, for 1-bit devices, volume is rather meaningless as you have only two volume states - nothing (off, 0) or full blast (on, 1). So instead, in PWM data each sample represents the time taken until the 1-bit output state will be toggled again. Sounds a bit confusing? Well, you can also think of PWM data as a sequence of frequencies. So, think about how a kickdrum sounds: It starts at a very high frequency, then quickly drops and ends with a somewhat longer low tone. So, as a PWM sample, we could create something like this:

```
;   >> high start and quick drop <<

db #80, #80, #70, #70, #60, #60, #60, #50
db #50, #50, #50, #40, #40, #40, #40, #40
db #30, #30, #30, #30, #30, #30, #20, #20
db #20, #20, #20, #20, #20, #20, #20, #10
db #10, #10, #10, #10, #10, #10, #10, #08
db #08, #08, #08, #08, #08, #08, #08, #08

;   >> slow low end <<
db #04, #04, #04, #04, #04, #04, #04, #04
db #04, #04, #04, #04, #04, #04, #04, #04
db #02, #02, #02, #02, #02, #02, #02, #02
db #02, #02, #02, #02, #02, #02, #02, #02
db #02, #02, #02, #02, #02, #02, #02, #02

;   >> end marker <<
db #00
```

Still confused? Well, luckily there's a utility that you can use to convert PCM to PWM. It's called pcm2pwm and can be downloaded here.

Now, how to play back this data? It couldn't be simpler. We need 3 variables:

- **data-pointer** - a pointer that points to the memory location of the PWM data. We'll use HL in our ZX Spectrum Z80 asm example.
- **counter** - a counter that is fed with the sample values. We'll use B.
- **state** - the output state. We'll use A' (the "shadow" accumulator).

Also, we need the ch-toggle constant as usual:

```
drums:
    ld a,0x10        ; state := on

drumloop:
    ex af,af'        ; counter := (data-pointer)
    ld b,(hl)
    xor a            ; IF counter == 0 THEN
    or b
    ret z            ; ... EXIT
    ex af,af'
innerloop:
    out (0xfe),a     ; OUTPUT state

    djnz innerloop   ; IF counter-- != 0 THEN GOTO innerloop

    inc hl           ; INCREMENT data-pointer
    xor 0x10         ; state := state XOR ch-toggle
    jr drumloop      ; GOTO drumloop
```

You can call this code inbetween notes, just like with the interrupting click drum method. However, this will lead to the usual problems - the drum sound needs to be very short, and you need to correct the main soundloop timer. A much better way to use PWM samples is to treat them like an extra channel, and trigger them within the soundloop alongside with the regular tone channels. The above code should be easy to adjust, so I'll leave that to you

7 VARIABLE PULSE WIDTH

Alright, if you've come this far, you should be able to write a pretty decent basic 1-bit sound routine. But the real fun of 1-bit coding has only started. From this point on, coding for 1-bit sounds becomes somewhat of an art form - you've got to use your creativity and imagination in order to build a routine that does something out of the ordinary.

I'm by no means an assembly expert, and don't understand half of what all these crazy beeper engines out there are doing. So I can only share those few tricks and techniques that I have so far discovered/reverse-engineered/been told about.

Ok, let's talk about variable pulse width. Varying the pulse width has a number of useful effects, most importantly the ability to produce more interesting timbres when used in conjunction with the pulse interleaving method. (In conjunction with PFM, it can be used to create volume envelopes, but this is not what this part of the tutorial is about.)

Imagine a classic pulse interleaving routine with 16-bit counters, as explained in part 4. The basic procedure for updating a channel's state and counters is:

```
counter := base + counter
IF carry THEN
  state := state XOR ch_toggle
ENDIF
OUTPUT state
```

This will output a square wave with a 50:50 duty cycle, because half of the time the output is 0, and the other half it's 1. Well, there is another of way of doing this:

```
       ld hl,nnnn    ; counter := base + counter
       add hl,bc
       ld b,h
       ld c,l
       ld a,h        ; IF counter < 8000h THEN
       cp 0x80
       ld a,0        ;   state := off
       jr nc,skip
;                    ; ELSE
       ld a,#10      ;   state := on
skip:                ; ENDIF
       out (#fe),a   ;    OUTPUT state
```

So, instead of waiting until the counter wraps from FFFFh to 0h, we now check if it has wrapped from FFFFh to 0h or from 7FFFh to 8000h. So in effect we change the state twice as often, but we will still get a 50:50 square wave. Now what happens if we compare against a value other than 8000h? You probably can guess: Yes, that will change our duty cycle. So, to get a 25:75 square wave for example, we'd compare against 4000h, for 12.5:87:5 we compare against 2000h, and so forth. Simple, right?

If only we wouldn't have to deal with that ugly conditional jump that ruins our timing. Well, in Z80 asm there's a handy trick. It is used in Shiru's Tritone, for example:

```
tritone:
ld hl,nnnn        ;do the counter math as usual
add hl,bc
ld b,h
ld c,l
ld a,h            ;compare against our chosen value
```

```
cp nn
sbc a,a          ; A will become 0 if there was no carry, else it becomes FFh
and 0x10         ; AND-ing 0x10 will leave A set to either 0 or 0x10,
;                        ;   depending the on previous result
out (0xfe),a
```

8 MORE SOUNDLOOP TWEAKS

In this part, I'll discuss two advanced tricks that you can use to open up new possibilities and further speed up your sound loops. I've learned these tricks from code by introspec and Alone Coder, respectively.

8.1 Accumulating Pin Pulses

The first trick applies to the PFM/pin pulse method of synthesis. First, let's take a look again at our PFM engine code from Part 5, and modify it to use 16-bit counters.

- **BC** = base frequency channel 1
- **IX** = freq. counter ch1
- **DE** = base freq. ch2
- **IY** = freq. counter ch2
- **HL** = timer

```
soundLoop:
       add ix,bc               ; update counter ch1
       sbc a,a                 ; A = #FF if carry, else A = 0
       and 0x10                ; A = #10 || A = 0
       out (0xfe),a            ; output state ch1

       add iy,de               ; same as above, but for ch2
       sbc a,a
       and 0x10
       out (0xfe),a

       dec hl                  ; decrement timer
       ld a,h
       or l
       jp z,soundLoop          ; and loop until timer == 0
```

Now, instead of outputting the pin pulses immediately after the counter updates, we can also "collect" them. This will potentially save some time in the sound loop and will give better sound, because the pin pulses will be longer.

In the following example, register A holds the number of pulses to output, and A' will hold 0x10.

```
soundLoop:
       add ix,bc               ; counter.ch1 := counter.ch1 + basefreq.ch1
       adc a,0                 ; IF carry, increment pulseCounter

       add iy,de               ; counter.ch2 := counter.ch2 + basefreq.ch2
       adc a,0                 ; IF carry, increment pulseCounter

       or a
       jp nz,outputOn
```

```
       out (#fe),a              ; OUTPUT state
       nop                      ; adjust timing
       nop
       nop

;                               ; now we can't use A to check the counter, hence...
       dec l                    ; decrement timer lo-byte
       jp z,soundLoop           ; and loop if != 0
; this is faster on average anyway, so use it whenever you can.
       dec h
       jp z,soundLoop           ; and loop until timer == 0

outputOn:
       ex af,af'
       out (#fe),a
       ex af,af
       dec a                    ; decrement pulseCounter

       dec l                    ; decrement timer lo-byte
       jp z,soundLoop           ; and loop if != 0
; this is faster on average anyway, so use it whenever you can.
       dec h
       jp z,soundLoop           ; and loop until timer == 0
```

Even better, you can use this trick to simulate different volume levels, by adding a number >1 to the pulse counter on carry. Just don't overdo it, because eventually the engine will overload, ie. it will take too long until the engine works through the "backlog" of pulses to output. This method is used in OctodeXL, btw.

8.2 Skipping Counter Updates

You have a great idea for a sound core, but just can't get it up to speed? Well, here's a trick you can use to make your loop faster. This trick is mostly relevant to pulse-interleaving engines with more than 2 channels.

The idea here is that you don't have to update all counters on each iteration of the sound loop. It is however important that you output all the states every time, and that the volume (read: time taken) of each of the channels is equal across loop iterations.

Here's a theoretical, not very optimized example.

- **DE** = base frequency ch1
- **IX** = counter ch1
- **H** = output state ch1
- **SP** = base frequency ch2
- **IY** = counter ch2
- **L** = output state ch2
- **B** = timer

```
       ld hl,0          ; initialize output states
       ld c,0xfe        ; port value, needed so we can use out (c),r command

soundLoop:              ; ---LOOP ITERATION 1---
       out (c),h        ; 12       ;OUTPUT state ch1
```

```
;                   ; ^ch2: 40
       add ix,de    ; 15      ;counter.ch1 := counter.ch1 + basefreq.ch1
       out (c),l    ; 12
;                   ; ^ch1: 27
       sbc a,a      ; 4       ;IF carry, toggle state ch1
       and 0x10     ; 7
       ld h,a       ; 4
       ld a,r       ; 9       ;adjust timing
       nop          ; 4

;                   ; ---LOOP ITERATION 2---
       out (c),h    ; 12
      ; ^ch2: 40
       add iy,sp    ; 15      ;counter.ch2 := counter.ch2 + basefreq.ch2
       out (c),l    ; 12
      ; ^ch1: 27
       sbc a,a      ; 4
       and #10      ; 7       ;IF carry, toggle state ch2
       ld l,a       ; 4
       djnz soundLoop ; 13    ;decrement timer and loop if !0
```

9 ACHIEVING SIMPLE PCM WITH WAVETABLE SYNTHESIS

The idea behind pulse-code modulated sound (PCM) is remarkably simple. A PCM waveform consists of a set of samples which describe the relative volume at a given time interval of constant length. (Note the terminology of "sample" in this context, which has nothing to with a sample as we know it from MOD/XM, for example). For playback, each sample is translated into a discrete voltage level, which is then amplified and ultimately sent to an output device, typically a loudspeaker. The samples are read and output sequentially at a constant rate until the end of the waveform has been reached.

When attempting this on a 1-bit device, we face the problem that we obviously can't output variable voltages. Instead we only have the choice between two levels, silence or "full blast". So how can we do it, then?

In order to understand how we can output PCM on a 1-bit device, let's first recap how Pulse Interleaving works. The underlying principle of P.I. is that we can keep the speaker cone in a floating state between full extension and contraction by changing the output state of our 1-bit port at a very fast rate, thanks to the inherent latency of the cone. So we're actually creating multiple volume levels. I'm sure you've realized by now that the same principle can be applied for PCM playback.

So, say we want to output a single PCM waveform at a constant pitch. All we need to do is *interpret the volume levels described by the samples as the amount of time we need to keep our 1-bit port switched on*. So we just create a loop of constant length, in which we

- read a sample
- switch the 1-bit port on for a the amount of time which corresponds to the sample volume
- switch the 1-bit port off for the remaining loop time
- check if we've reached the end of the waveform, and loop if we haven't.

That's all - on we go with the next sample, rinse and repeat until the entire waveform has been played.

Loop duration is a critical parameter here, of course. We can't make our loop too long, or else the "floating speaker state" trick won't work. It seems that a loop time of around 1/15000 seconds is the absolute maximum, but ideally you should do it a bit faster than that.

With common PCM WAVs, we'll run into a problem at this point. An 8-bit PCM WAV has samples which can take 256 different volume levels, take the more popular 16-bit ones and you've already got 65536 levels. How are we supposed to control timing that precisely in our loop? 1/15000 seconds corresponds to around 233 cycles on the ZX Spectrum. The fastest output command - OUT (n),A - takes 11 cycles, which means we can squeeze at most 21 of those into the loop - and that's not taking into account all the tasks we need to perform besides outputting. So how do we output 256 or even 65536 levels? The answer is: We don't. Instead, we'll reduce the sample depth (that is, the number of possible volume levels) to a suitable level. This will obviously degrade sound quality, but hey, it's better than nothing.

As far as the Spectrum is concerned, 10 levels seems to be a convenient choice. You might be able to do more with clever code (or on a faster machine), but for the purpose of this tutorial, let's keep it at 10. That is, if we want to output just a single waveform. But of course we want to mix multiple waveforms at variable pitches, let's say two of them. In this case, our source PCM waveforms should have 5 volume levels.

As you might have already guessed, we'll need to develop our own PCM data format to encode these 5 levels. How this format will look like depends on your sound loop code as well as the device you're targetting - anything goes to make things as fast as possible. On the Spectrum, we may take two things into account:

- bit 4 sets the output state (let's ignore the details for now...)
- we have a fast command available for rotating the accumulator.

So, our samples bytes might look like this:

```
volume  binary    hex
level   76543210
-----------------------
  0%    00000000  0x00
 25%    00010000  0x10
 50%    00011000  0x18
 75%    00011100  0x1c
100%    00011110  0x1e
```

This reasoning behind this may not be self-evident, but it'll become clear when we look at a possible sound loop.

Unfortunately, this custom PCM format still won't allow us to create a sound loop that is fast enough, so let's apply another restriction - use waveforms with a fixed length of 256 byte-sized samples. You'll see in a moment why this comes in handy.

Our sound loop might look like this:

```
sound:
  set up sample pointer channel 1             |  ld bc,waveform1
  set base frequency ch1                      |  ld de,noteval1
  clear add counter ch1                       |  ld hl,0
       |  exx
  set up sample pointer channel 2             |  ld bc,waveform2
  set base frequency ch2                      |  ld de,noteval2
  clear add counter ch2                       |  ld hl,0
  set timer                                   |  ld ix,0
```

```
loop:                                          |
  load channel 1 sample byte to accumulator    |  ld a,(bc)
  output accu to beeper                        |  out (#fe),a
  rotate left accumulator                      |  rlca
  output accu to beeper                        |  out (#fe),a
  rotate left accumulator                      |  rlca
  output accu to beeper                        |  out (#fe),a
  rotate left accumulator                      |  rlca
  output accu to beeper                        |  out (#fe),a
  add base frequency ch1 to counter ch1        |  add hl,de
    IF counter overflows,                      |  adc a,0 \ add a,c
    advance sample pointer ch1                 |  ld c,a \ exx
  ...                                          |  exx
  load channel 2 sample byte to accumulator    |  ld a,(bc)
  output accu to beeper                        |  out (#fe),a
  rotate left accumulator                      |  rlca
  output accu to beeper                        |  out (#fe),a
  rotate left accumulator                      |  rlca
  output accu to beeper                        |  out (#fe),a
  rotate left accumulator                      |  rlca
  output accu to beeper                        |  out (#fe),a
  add base frequency ch2 to counter ch2        |  add hl,de
    IF counter overflows,                      |  adc a,0 \ add a,c
      advance sample pointer ch2               | ld c,a

  decrement timer and loop if not 0            |  dec iy \ ld a,iyh
  ...                                          |  or iyl
  ...                                          |  jp nz,loop
```

Now you also see why limiting waveforms to 256 bytes is useful - this way, we can loop through them without ever having to reset the sample pointer, which of course saves time.

However, there's a whole array of problems with this code. First of all, it's still quite slow - 218 cycles. Secondly, you can see that the last output from each channel last significantly longer than the first 3. A bit of difference in length is actually not a big problem, but in this case, the last frame is 3 times longer - that's simply too much. Thirdly and most critically, I/O contention has not been taken care of (this mainly concerns the Speccy, of course).

If you've followed the discussion in this thread, you'll have noticed that I normally don't pay as much attention to I/O contention as other coders, but in this case, aligning the outputs to 8 t-state limits does make a huge difference. I'll let you figure this out on your own though. Check my wtfx code if you need further inspiration.

I will tell you one important trick for speeding up the sound loop though. Credits for this one go to sorchard from World of Dragon.

In the above sample, we're actually using 24 bit frequency resolution, since we're keeping track of the overflow from adding our 16-bit counters. But 16 bits are quite enough to generate a sufficiently accurate 7-8 octave scale. So in the above example, instead of doing "adc a,0 \ add a,c \ ld c,a" to update the sample counter, you could simply do "ld c,h", saving a whopping 22 cycles in total. The high byte of our add counter thus becomes the low byte of our sample pointer. The downside of this is that our waveforms need to be simple - e.g. just one iteration of a basic wave (triangle, saw, square, etc.). It's less of a problem than it sounds though, as you

won't be creating really complex waveforms in 256 bytes anyway. And for a kick drum or noise, you can simply use a frequency value <256, making sure that you step through every sample in the waveform.

And that's all for now, hope you find the information useful, and as always, let me know if you find any errors or have any further suggestions/ideas.

10 SOUND TRICKS – NOISE, PHASING, SID SOUND, EARTH SHAKER, DUTY MODULATION

Digital/PCM sound is a powerful and flexible tool, but unfortunately it tends to consume a lot of RAM. So in this part of the tutorial, let's go back to the good old Pulse Interleaving technique, and talk about various tricks that can be used to spice up its sound.

10.1 Noise

Historically speaking, 1-bit routines have always been lacking in terms of percussive effects. However, converting a tone generator into a simple noise generator is surprisingly simple, and costs a mere 8 cycles on Z80-based systems. Consider the usual way we generate square wave tones:

```
noise:
       add hl,de           ; add base frequency divider (DE) to channel accumulator (HL)
       ld a,h              ; grab hi-byte of channel accumulator
       cp DUTY             ; compare against duty threshold value
       sbc a,a             ; set A to 0 or 0xFF depending on result
       out (0xfe),a        ; output A to beeper port
```

Now, in order to generate noise instead of tones, one would obviously need to randomize the value held by the channel accumulator. But pseudo-random number generators are slow, so how do we do it? The answer is to simply reduce the quality of the PRNG as much as possible. Believe it or not, adding a single instruction:

```
noise:
       add hl,de           ; add base frequency divider (DE) to channel accumulator (HL)

       rlc h               ; << ADDED

       ld a,h              ; grab hi-byte of channel accumulator
       cp DUTY             ; compare against duty threshold value
       sbc a,a             ; set A to 0 or 0xFF depending on result
       out (0xfe),a        ; output A to beeper port;
```

after the ADD HL,DE operation will provide enough entropy to create a convincing illusion of white noise. But it will do so only if it is fed a suitable

frequency divider as a seed. I usually use a fixed value of 0x2174 in my routines. Other values are possible of course, and may give slightly different results, though most values will just generate nasty glitch sounds instead of noise.

There's a nice side effect that you get for free - you can create the illusion of controlling the volume of the noise by changing the duty threshold. Changing the pitch of the noise however is much more difficult, and requires the use of additional counters. I'm still looking for an efficient way of doing it, if you have any ideas please let me know.

Last note, you can of course also rotate the hi-byte of frequency divider instead of the accu. The result of that however is almost guaranteed to be a glitch sound rather than noise.

10.2 Phasing

The Phasing technique was developed by Shiru, and is used to generate the signature sound of his Phaser1-3 engines. It comes at a rather heavy cost in terms of cycle count and register usage, but it's power and flexibility undoubtedly outweigh those drawbacks.

For regular tone generation, we use a single oscillator to generate the square wave (represented by the add hl,de operation). The main idea of Phasing, on the other hand, is to use two oscillators, and mix their outputs into a single signal. The mixing can be done via a binary XOR of the two oscillator outputs (the method used in Phaser1), or via a binary OR or AND (added in Phaser2/3).

```
phasing:
   add hl,de           ; OSC 1: add base freq divider (DE) to channel accu (HL) as usual
   ld a,h              ; grab hi-byte of channel accumulator
   cp DUTY1            ; compare against duty threshold value
   sbc a,a             ; set A to 0 or 0xFF depending on result
   ld b,a              ; preserve result in B

   exx                 ; shadow register set, yay
   add hl,de           ; OSC 2: exactly the same operation as above
   ld a,h              ; grab hi-byte of channel accumulator
   cp DUTY2            ; compare against duty threshold value
   sbc a,a             ; set A to 0 or 0xFF depending on result
   exx                 ; back to primary register set

   xor b               ; combine output of OSC 1 and 2. (xor|or|and)
   out (0xfe),a        ; output A to beeper port
```

As you can see, this method offers a wide range of parameters that affect timbre. The most important one, from which the technique derives its name, is the phase offset between the two oscillators. To make use of this feature, simply initialize the OSC1 accu to another value than the initial value of the OSC2 accu, eg. initialize HL to 0 and HL' to a non-zero value. Especially in conjunction with a slight offset between the OSC1 and OSC2 base dividers, some surprisingly complex timbres can be produced.

Side note: By using a binary OR to mix the signal and keeping the duty thresholds down to a reasonable level, the two oscillators can be used as independant tone generators. This method is used to mix channels in Squeeker and derived engines.

10.3 SID Sound

This effect, which derives its name from the key sound that can be heard in many of the early SID tunes, is formed by a simple duty cycle sweep. The velocity of the sweep is in sync with the frequency of the tone generator. Basically, every time the channel accumulator overflows, the duty threshold is increased or decreased. As with noise, this is trivial to pull off and costs only a few cycles. Using the standard tone generation procedure, we can implement it as follows:

```
sid_sound:
      add hl,de          ; add base frequency divider (DE) to channel accumulator (HL)
      sbc a,a            ; set A to 0 or 0xFF depending on result
```

```
      add a,c             ; add duty threshold (C)
      ld c,a              ; update duty threshold value (C = C - 1 if add hl,de carried)
      cp h                ; compare duty threshold value against hi-byte of channel accu
      sbc a,a             ; set A to 0 or 0xFF depending on result
      out (0xfe),a        ; output A
```

As you can see, this operation costs a mere 4 cycles compared to the standard procedure without duty cycle sweep.

10.4 Earth Shaker

This effect is named after the game Earth Shaker, which used a rather unusual sound routine with two semi-independant tone channels, written by Michael Batty. As an actual method of generating multi-channel sound, it is of limited practicality, but it can be applied as an effect to regular Pulse Interleaving at a minimal cost. The core concept here is to continually modulate the duty threshold within the sound loop. Depending on the ratio of the duty cycle change vs the oscillator speed, the result can be a nice chord, phatness, or - in most cases - gruesome disharmony that will strike fear in the hearts of even the most accustomed 1-bit connaisseurs. A simple implementation, as used in HoustonTracker 2 for example, looks like this:

```
earth:
       add hl,de           ; add base frequency divider (DE) to channel accumulator (HL)
       ld a,c              ; load duty threshold (C)
       add a,DUTY_MOD      ; add duty threshold modifier
       ld c,a              ; store new duty threshold
       cp h                ; compare duty threshold value against hi-byte of channel accu
       sbc a,a             ; set A to 0 or 0xFF depending on result
       out (0xfe),a        ; output A to beeper port
```

10.5 Duty Modulation

The aforementioned SID sound and Earth Shaker effects are actually basic implementations of a family of effects that may best be described as "Duty Modulation". As a first step into the world of Duty Modulation, let's take the Earth Shaker effect and modify it to change the duty threshold in sync with the main oscillator.

```
duty_modulation:
       add hl,de           ; add base frequency divider (DE) to channel accumulator (HL)
       sbc a,a             ; set A to 0 or 0xFF depending on result
       and DUTY_MOD        ; set A to 0 or DUTY_MOD
       xor c               ; XOR with current duty threshold (C)
       ld c,a              ; store new duty threshold
       cp h                ; compare duty threshold value against hi-byte of channel accu
       sbc a,a             ; set A to 0 or 0xFF depending on result
       out (0xfe),a        ; output A to beeper port
```

By syncing the modulation in this way, the nasty glitches of the Earth Shaker effect can be avoided entirely (but also, no chords will be produced). Instead, we can now control harmonic components that share an octave relation with the base note. In other words, we can amplify over- and undertones at will, as long as they are a multiple of 12 half-tones away from the main note.

Things can be pushed even further by decoupling the sync and using a second oscillator to time the duty threshold updates.

```
double_duty:
       exx
```

```
        add hl,de           ; independant oscillator for timed duty threshold updates
        exx
        sbc a,a             ; set A to 0 or 0xFF depending on result
        and DUTY_MOD        ; set A to 0 or DUTY_MOD
        xor c               ; XOR with current duty threshold (C)
        ld c,a              ; store new duty threshold

        add hl,de           ; add base frequency divider (DE) to channel accumulator (HL)
        cp h                ; compare duty threshold value against hi-byte of channel accu
        sbc a,a             ; set A to 0 or 0xFF depending on result
        out (#fe),a         ; output A to beeper port
```

This way, we can create the octave effects from the previous example (by setting the "duty" oscillator to the same value as the main tone oscillator), as well as Earth Shaker style chords, while also gaining better control over the latter. Additionally, some interesting slow-running timbre changes can be achieved by setting the duty oscillator to a frequency near (but not equal to) the main oscillator.

The usefulness of this approach might seem a bit questionable considering the hefty cost in CPU cycles and register usage. However, the required code is almost the same as the one used for the Phasing technique, so with a tiny amount of self-modifying code, it can be implemented in a Phaser style engine at virtually no extra cost.

There's also an added bonus when combining this technique with the noise generator explained above. By setting the duty threshold to the same value as the duty modifier, the duty oscillator can be used as a tone generator, meaning you can actually mix noise and tone on the same channel!

That's all for this time. If you know of any other cool tricks please post them here!

11 SYNTHESIZING BASIC WAVEFORMS: RECTANGLE, TRIANGLE, SAW

In this chapter, I'm going to explain how to synthesize different waveforms without the use of samples or wavetables.

For generating waveforms other than a rectangle/pulse on a 1-bit output, we need to be able to output multiple volume levels. In part 10, we have looked at some methods for outputting PCM samples and wavetables. We concluded that in the 1-bit domain, time is directly related to volume. The longer we keep our 1-bit output "on" within a fixed-length frame, the higher the volume produced by the speaker cone will be. We can use this knowledge to write a very efficient rendering loop that will generate 8 volume levels with just 3 output commands:

```
calculate 3-bit volume
output (volume & 1) for t cycles
output (volume & 2) for 2t cycles
output (volume & 4) for 4t cycles and loop
```

As you can see, the trick here is to double the amount of cycles taken after each consecutive output command in the loop. An implementation of this for the ZX Spectrum beeper could look like this:

```
      ld c,0xfe
      ld hl,0
      ld de,frequency_divider
loop
```

```
      add hl,de         ;11                  ; update frequency counter as usual
      ;...              ;y                   ; do some magic to calculate 3-bit volume
      ;...              ;z                   ; put it in bit 4-6 of register A
      ; so now bit 4 of A = volume & 1
      out (c),a         ;12: x+10+11+y+z=64  ; output to beeper
      rrca              ;4                   ; now bit 4 of A = volume & 2
      out (c),a         ;12: 4+12=16         ; output
      ds 4              ;16                  ; timing
      rrca              ;4                   ; now bit 4 of A = volume & 4
      out (c),a         ;12: 16+4+12=32      ; output
      ;...              ;x                   ; update timer etc.
      jp loop           ;10                  ; loop
```

If you count the cycles, you'll notice that this loop takes exactly 112 cycles. Which means we can easily add a second channel in the same manner, which brings the total cycle count to 224 - perfect for a ZX beeper routine. Side note: If necessary, you can cheat a little and reduce the 64-cycle output to 56 cycles, without much impact on the sound.

Anyway, we will use this framework as the basis for our waveform generation. So let's talk about the "magic" part.

The easiest of the basic waveforms is the saw wave. How so, you may ask? Well, the saw wave is actually right in front of your nose. Look at the first command in the sound loop - ADD HL,DE. Say we set the frequency divider in DE to 0x100. What happens to the H register? It is incremented by 1 each sound loop iteration, before wrapping around to 0 eventually. Ok, by now you might have guessed where this is going. If you haven't, then plot it out on a piece of paper - the value of H goes on the y-axis, and the number of loop iterations goes on the x-axis. Any questions? As you can see, our saw wave is actually generated for free while we update our frequency counter (thanks to Shiru for pointing this out to me). We just need to put it into A, and rotate once to get it into the right position.

```
; rotate the bit
      add hl,de         ;update frequency counter
      ld a,h            ;now 3-bit volume is in bit 5-7 of A
      rrca              ;now it's in bit 4-6
      out (c),a         ;output as above
      ...
```

Doing a triangle wave is a little more tricky. In fact, being the lousy mathematician that I am, it took me quite a while to figure this out. Ok, here's how it's done. We've already got the first half of our triangle wave done - it's the same as the saw wave. The second half is where the trouble starts - instead of increasing the volume further as we do for the saw wave, we want to decrease it again. So we could do something ugly like:

```
      add hl,de
      ld a,h            ;check if we've passed the half-way point of the saw
      rla               ;aka H >= 0x80
      jp c,invert_volume
      ...
reentry
      rrca
      out (c),a
      ...
      jp loop

invert_volume:
```

```
      ;A = -H
```

There's a more elegant way that does the same thing without the need for conditional jumps.

```
; no conditional jumps
      add     hl,de
      ld a,h
      rla
      sbc a,a          ;if h >= 0x80, A = 0xff, else A = 0
      xor h            ;0 xor H = H, 0xff xor H = -H - 1
      out (c),a        ;result is already in bit 4-6, no need to rotate
      ...
```

We can simply ignore the off-by-one error on H >= 0x80, since we don't care about the lower 4 bits anyway.

Last but not least, a word about rectangle waves. Of course, rectangle waves happen naturally on a 1-bit output, unless you force it to do something else. Which we are doing in this case, so how do we get things back to "normal"? Well, to get a square wave, we simply have to remove the XOR H from the previous code example. Which means that with just two bytes of self-modifying code, we can create a routine that will render saw, triangle, or square waves on demand:

```
; geenrator
      add hl,de
      ld a,h
      rla

      ; saw    |   tri       |   rect
      ;--------------------------------
      rra     |   sbc a,a   |   sbc a,a
      rrca    |   xor h     |   nop

      out (c),a
      ...
```

You'll notice that even with timer updates, register swapping, etc. you'll still have some free cycles left. Which should, of course, be put to some good use - see part 11 if you need some inspiration.