What is Arkos Tracker?

Arkos Tracker is a music software specialised on retro-computers such **Amstrad CPC**, **ZX Spectrum**, **Atari ST** and so on, which all share the same soundchip, the **AY-3-8912** and **YM-2149F**.

You can compose music on any modern desktop computer, then transfer the music (or sound effects) on the old computer thanks to the tailor-made assembler players.

Arkos Tracker has been already used in **hundreds** of games and demos! And now it's your turn!

Features

- **Cross-platform** (Windows, Linux, Mac).
- Open source.
- Manage both standard **PSG** sound and **samples** without any distinction!
- Support of **AY-3-8912** and **YM2149F**.
- **Unlimited PSG count**! You can create standard 3-channel songs, or 6, 9, 32 or more if you want!
- **4 columns of effects** per channel, including Arpeggio (via a table or inline value), Pitch, slide, glide, change Arpeggio/Pitch/Instrument speed, etc.
- Import from MIDI, AKS (Arkos Tracker 1 and 2), SKS (STarKos), 128 (BSC's Soundtrakker), WYZ (Wyz Tracker), MOD, VT2 (Vortex Tracker 2), CHP (Chip'n'sfx).
- **Several players**, from versatile to extremely fast, or slower but less memory-consuming.
- Support of **specific hardware**: PlayCity (CPC), Spectrum TurboSound, SpecNext, MSX Darky and MSX FPGA (AKY player only).
- **Real-time communication** with your hardware possible, to listen or compose just like on the real thing!
- **Sound effect support**, shared among songs.
- **ROM players** available: no more auto-modifying code, a small buffer is used.
- Exports can be **assembly sources** or binaries: it makes it easy to integrate the songs in your production.
- Sources can be **converted** to any assembler.
- Each PSG can have its own frequency (including custom ones).
- Replay rate from 12Hz to 300Hz, or custom.
- A song is composed of sub-songs, all sharing the same instruments. Helps you save even more memory!

All the limitations of past AY/YM trackers are blown to pieces with Arkos Tracker!

Understanding the PSG

Using Arkos Tracker efficiently requires a minimum knowledge of our beloved AY-3-8912/YM-2149F. We are going to have a quick overview of their capabilities, with sounds, for a better understanding.

As a simplification, most of this documentation will refer to the AY, but all the explanations also concern the YM. The difference between the two will be explained below.

Overview

The channels

The AY/YM are an old Programmable Sound Generator (PSG) which can produce sound in a rather limited, yet expressive way.

This PSG has **3 channels**. This means we can produce three sounds at the same time. The output may be **mono** or **stereo** depending on the hardware. The CPC has only one speaker, but also a stereo output for plugging an amplified system. The CPC Plus has two speakers, the Atari STF only one (arh arh).

Waves, amplitudes and periods

By default, the AY generates **square waves**. This is the most basic, saturated sound you can find.

0:00 / 0:02

Listening to three square waves is quite tiring, but fortunately, each wave has its own **amplitude**, so changing it in real-time allows creating more natural sounds.

0:00 / 0:00

There are 16 amplitude steps, from 0 (no sound) to 15 (full amplitude). The amplitude curve is **logarithmic**: this means that the amplitude difference between amplitude 10 and 11 is not as important as the difference between amplitude 14 and 15.

In order to play music, one must play notes. Each channel can be given its own frequency. The little example above, on top of decaying the amplitude, also changes their frequency over time.

Important: the PSG does not work with **frequencies** (440 Hz for example), but with **periods**. Periods are the invert of frequencies. The highest the frequency, the lowest the period. Mathematically speaking: period = 1 / frequency. However, we won't be using maths here so don't worry if you don't understand this. Just be aware that periods will be used, and thus by *decreasing* the period, the pitch of a sound will actually *increase*. For now on, we will talk about periods, not frequencies.

Noise

What about producing some drums? Well the PSG got you covered: it has **one noise generator**. It can be coupled to any of the three channels, or two of them, or even the three of them. The period of the noise varies thanks to a value going from 1 to 31 (0 producing the same result as 1, 1 being a very high-pitched noise, 31 a low-pitched one). Here is an example of the noise going from 0 to 31.

0:00 / 0:01

Noise can be used alone as seen above, or mixed with a square wave:

0:00 / 0:02

I told you there was only one noise generator. If you apply the noise on two channels, both will use the same noise value. This is quite limiting, which is why, most of the time, you will not use two noises at the same time. It simply sounds ugly. Forget your dreams of simulating a hihat (noise to 1) and big explosion sound (noise to 31) at the same time.

The hardware envelope

All that is above is enough to make 99.9% of the game music from the 80's. You may want to go one step beyond and use another neat feature called "the hardware envelope", which is very useful to produce peculiar sounds, mostly used in basses, but which can have great effects in melodies or for special effects.

As you have seen, in order to have more expressive sounds, you would use the amplitude to create attack and decay to simulate real (or not) instruments. The hardware envelope allows you to do it automatically.

One envelope to rule them all

Note the use of the singular when I talk about hardware envelope. Just like there is only one single noise generator, **there is only one single** hardware envelope generator. And just like noise, it can be used on one, two, or the three channels at once. But just like the noise, it will probably sound ugly if you try to use it in more than one channel at the time.

The shapes

Two parameters defines the hardware envelope:

- Its shape (going up, or down, cycling or not)
- Its period: how fast it goes.

There are 8 shapes available. Here are they, using a high period for you to hear them, and with the software wave on.

Shape 8: sawtooth from 15 to 0, loops.

0:00 / 0:02

Shape 9: from 15 to 0, loops at 0.

0:00 / 0:00

Shape 10 (0xA): triangle (15 to 0 to 15 and loops).

0:00 / 0:03

Shape 11 (0xB): 15 to 0, loops to 15.

0:00 / 0:01

Shape 12 (0xC): sawtooth from 0 to 15, loops (opposite to 8).

0:00 / 0:02

Shape 13 (0xD): 0 to 15, loops to 15.

0:00 / 0:01

Shape 14 (0xE): triangle (0 to 15 to 0 and loops) (opposite to 0xA).

0:00 / 0:03

Shape 15 (0xF): 0 to 15, loops to 0.

0:00 / 0:01

Some important remarks must be made :

- Some shapes seem to loop endlessly, some to stop after one cycle. However, they actually all loop on their last cycle.
- Whatever the shape is, the amplitude **always** goes between 0 and 15 (or the opposite). You can't ask a shape to go from 14 to 6 and then cycle or stop.

The latter limitation makes the hardware envelope as though it is useless. Why bother using an envelope when I can simply not use them but change the amplitude at will in real-time, choosing the exact values I want? In a sense, you are right, but the real interest is explained below.

Speed it up!

So hardware envelopes are limited and boring. Now let's try something. Let's play a sound using a cycling hardware envelope (like 8) and progressively decrease the period of the hardware envelope. In this example, the software envelope is fixed to a certain period and won't change.

0:00 / 0:11

Did you hear that? Isn’t it awesome? By using a low period of the hardware envelope, a whole new sound is created from a boring square wave. Note that most of the sample sounded like rubbish, up to the end, where it sounded right. Why? This is explained below.

Synchronization

Why did it sound right? Because the period of the (square) software wave is proportional to the one of the hardware envelope. A good result is to have the latter being 8, 16 or 32 times (a power of 2) faster than the square wave:

Hardware period 8 times faster than the software period, shape 8.

0:00 / 0:03

Hardware period 16 times faster than the software period, shape 8.

0:00 / 0:03

Hardware period 8 times faster than the software period, shape 10.

0:00 / 0:03

Hardware period 16 times faster than the software period, shape 10.

0:00 / 0:03

So this sounds great, but you will probably very quickly encounter a limitation: the higher the pitch of the sound, the less accurate the periods are. Plus, the hardware envelope is more accurate than the software. So there *will* be times where envelopes will be desynchronized. Most of the time, with low sounds, it won’t sound too disgracious. But go higher and…

0:00 / 0:03

The faster the internal clock of the AY is, the more accurate this will be. Which explains why some Atari ST music converted to the CPC can have its hardware sounds sound crappy: the Atari ST has a 2 mHz YM, the CPC a 1 mHz AY.

Also, don't worry about the ratio calculations we talked about earlier. Unless you specifically want to, everything is automatically calculated by AT! You don't have to worry about the technicalities, only fiddle with some parameters and have fun with what you hear.

AY or YM?

The AY is used on many computers (Amstrad CPC, ZX Spectrum, MSX, etc.). The YM is mostly used on the Atari ST and MSX 2 and is almost exactly the same as AY (from which it copied its design). AT fully supports both.
There are two subtle differences between the AY and YM:

- The logarithm curves are slightly different. The YM sounds "smoother" because the steps between the highest volumes aren't as "steep".
- The hardware envelopes on the YM have double steps, which means that intermediate volumes are used when ramping up or down. This once again improves the "smoothness" of the hardware sounds. This is especially noticeable when using "hardware only" sounds, where the hardware envelope is distinctly heard. However, these more accurate volumes **are only used** when using hardware envelope (you don't have control over them). You **can not** use them with the usual basic volumes… Too bad.

Samples

You may wonder how samples can be played on such limited chips. Well, depending on the hardware, it may be very easy to not-so-easy. But the basic is as this: very quickly, the amplitude of one (or more!) channel(s) is changed very quickly (8000 times per seconds for a 8 kHz sound), thus producing a richer sound than the PSG can normally produce by itself. Hardware like Atari ST have timers allowing to do that "in the background". But on more limited hardware (like 8-bit computers), it is up to the coder to perform these changes: synchronization in itself is not hard to do, but things can get very complex if you want the computer to perform other actions while playing the sample.

As of now, except for the MOD player, no player in AT handle samples, but a digi-drum player may be done if people ask me to.

SIDs

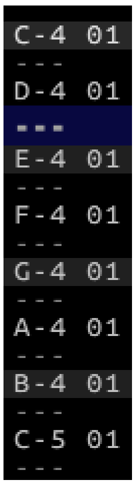
SIDs have been created on Atari ST to emulate (a bit) the fantastic sound chip of the C64. Without the use of the Atari ST timers, they are hard to do, and the needed accuracy will require 100% of the CPU, which make their use pretty limited (and I personally think the resulting sound is not so great, making the trouble to create such complex player rather useless). They are also many ways to produce SIDs (ST-SID, Sid samples, etc.). As of now, SIDs are not supported by AT.

Positions, patterns, tracks and subsongs

This tutorial explains the concepts about positions, patterns, tracks and subsongs in Arkos Tracker. If you've used a tracker before, all these will sound familiar, so you can skip this chapter. However, users of STarKos, AT1 and 2 **should** read this: things have changed with AT3 (simplified actually. A good thing, right?).

Tracks

Tracks are the building blocks of your song. Consider a track a musical sheet for one musician. A track contains notes, the instrument to use, and possibly effects. The following track…



```
C-4 01
---
D-4 01
---
E-4 01
---
F-4 01
---
G-4 01
---
A-4 01
---
B-4 01
---
C-5 01
---
```

… sounds like this:

0:00 / 0:01

Each symbol (C-4, 01, etc.) are explained thoroughly in the pattern viewer page, so let's not worry about that for now. Simply accepts that notes are written in tracks.

Tracks are limited, on purpose:

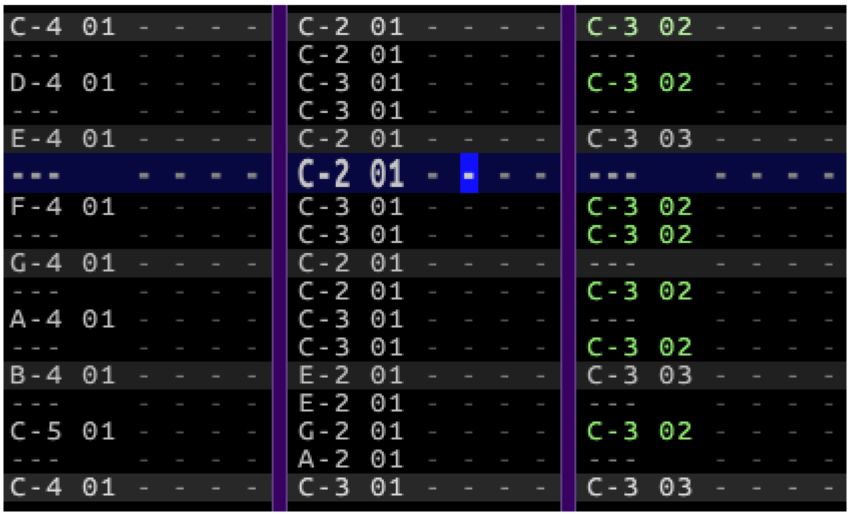
- First, in size. Basically, they can contain from 1 to 128 notes maximum.
- Then, tracks contain one note at the time (they are monophonic, so to speak).

Building a whole song with such limitations is impossible, which is why tracks are only used as building blocks. Think of a track as the smallest part of a Lego construction. Tracks are actually contained in…

Patterns

Patterns are simply an array of tracks. How many depends on your hardware: 3 on most (Amstrad CPC, Spectrum, MSX, Atari ST etc.), simply because that's what the AY/YM can handle. AT allows using custom chips such as PlayCity, Spectrum Next, Darky, increasing the track count per pattern to 6 or 9 (or even more if you want to, as AT itself has no limitation).

Now with 3 tracks, we have polyphony!



```
C-4 01 - - - -   C-2 01 - - - -   C-3 02 - - - -
---    - - - -   C-2 01 - - - -   ---    - - - -
D-4 01 - - - -   C-3 01 - - - -   C-3 02 - - - -
---    - - - -   C-3 01 - - - -   ---    - - - -
E-4 01 - - - -   C-2 01 - - - -   C-3 03 - - - -
---    - - - -   C-2 01 - - - -   ---    - - - -
F-4 01 - - - -   C-3 01 - - - -   C-3 02 - - - -
---    - - - -   C-3 01 - - - -   C-3 02 - - - -
G-4 01 - - - -   C-2 01 - - - -   ---    - - - -
---    - - - -   C-2 01 - - - -   C-3 02 - - - -
A-4 01 - - - -   C-3 01 - - - -   ---    - - - -
---    - - - -   C-3 01 - - - -   C-3 02 - - - -
B-4 01 - - - -   E-2 01 - - - -   C-3 03 - - - -
---    - - - -   E-2 01 - - - -   ---    - - - -
C-5 01 - - - -   G-2 01 - - - -   C-3 02 - - - -
---    - - - -   A-2 01 - - - -   ---    - - - -
C-4 01 - - - -   C-3 01 - - - -   C-3 03 - - - -
```

Listen to this beauty:

0:00 / 0:02

The first track (on the left) is the melody, the second is the bass, the third is the drums. However *magnificent* this music is, it's a bit short. Patterns too have limitations:

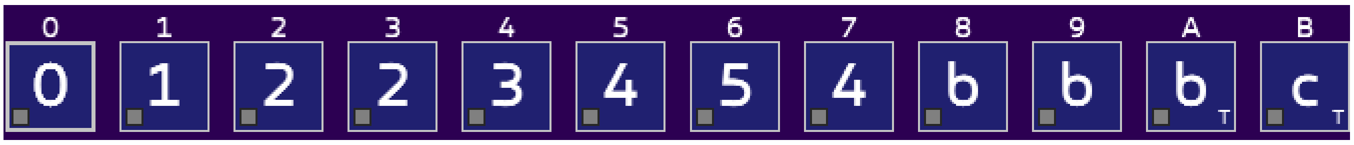
- They have the same track amount as your target hardware allows it.
- Since they just contain tracks, patterns can only be 1 to 128 notes high too.

Must our songs be this short? Of course not. Now positions come to the rescue…

Positions

A song is actually only a sequence of positions. A position is a reference to a pattern, plus other smallish data. Positions are referred to via an increasing number, starting at 0. Your song will start at position 0, then 1, 2 and so on.

How are patterns referred to? Via a number too, starting at 0. Let me show you a bit how the sequencing looks like:



```
0  1  2  3  4  5  6  7  8  9  A  B
0  1  2  2  3  4  5  4  b  b  b  c
                              T  T
```

The top numbers are the positions. As you can see, they are increasingly numbered. Don't bother remembering the position numbers, this would be quite useless.

The big numbers are the pattern numbers. So position 0 refers to pattern 1, position 1 to pattern 1, position 2 to pattern 2, position 3 to pattern 2 once again. This is interesting. Position 2 and 3 highlight the concept of *repetitions*. Patterns are exactly made for that: it is a sequence of notes that may be reused a certain amount of times without you having to enter the notes again, which would be time consuming and a waste of memory.

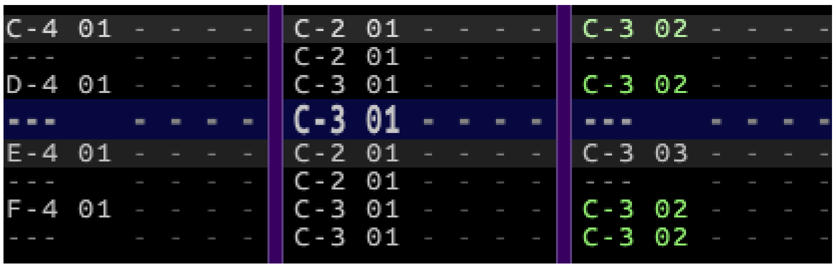
You can see this concept even further in the song: position 5 and 7 refer to the same pattern 4. And that lazy musician ended the song with three times the pattern B (this is hexadecimal for 11).

Positions have an interesting set of data, asides the pattern number:

Height

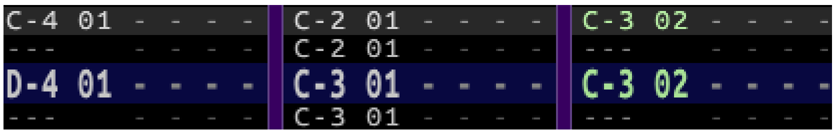
We saw earlier that tracks, and thus patterns, could contain 1 to 128 notes. Positions are actually the ones that tell how long the patterns are. This is a handy feature, because it allows to use the patterns multiple times with different length. Let's go back to our polyphonic example before. Imagine that I want the first position to play only 8 lines of it, the second position only 4, and a third position, all of it (so, 3 positions but only one pattern). This would *look* like:

Position 0 (8 lines only):



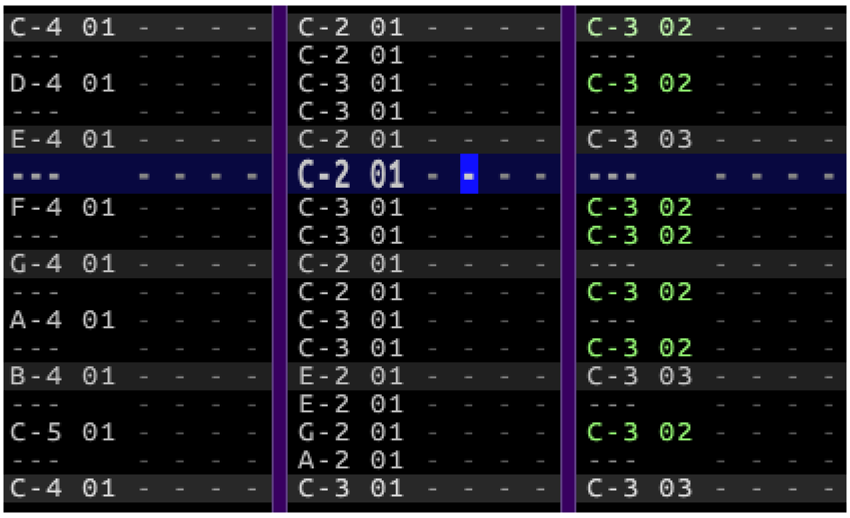
```
C-4 01 - - - - | C-2 01 - - - - | C-3 02 - - - -
---    - - - - | C-2 01 - - - - | ---    - - - -
D-4 01 - - - - | C-3 01 - - - - | C-3 02 - - - -
---    - - - - | C-3 01 - - - - | ---    - - - -
E-4 01 - - - - | C-2 01 - - - - | C-3 03 - - - -
---    - - - - | C-2 01 - - - - | ---    - - - -
F-4 01 - - - - | C-3 01 - - - - | C-3 02 - - - -
---    - - - - | C-3 01 - - - - | C-3 02 - - - -
```

Position 1 (4 lines only):



```
C-4 01 - - - -  C-2 01 - - - -  C-3 02 - - - -
---    - - - -  C-2 01 - - - -  ---    - - - -
D-4 01 - - - -  C-3 01 - - - -  C-3 02 - - - -
---    - - - -  C-3 01 - - - -  ---    - - - -
```

Position 2 (17 lines):



```
C-4 01 - - - -  C-2 01 - - - -  C-3 02 - - - -
---    - - - -  C-2 01 - - - -  ---    - - - -
D-4 01 - - - -  C-3 01 - - - -  C-3 02 - - - -
---    - - - -  C-3 01 - - - -  ---    - - - -
E-4 01 - - - -  C-2 01 - - - -  C-3 03 - - - -
---    - - - -  C-2 01 - - - -  ---    - - - -
F-4 01 - - - -  C-3 01 - - - -  C-3 02 - - - -
---    - - - -  C-3 01 - - - -  C-3 02 - - - -
G-4 01 - - - -  C-2 01 - - - -  ---    - - - -
---    - - - -  C-2 01 - - - -  C-3 02 - - - -
A-4 01 - - - -  C-3 01 - - - -  ---    - - - -
---    - - - -  C-3 01 - - - -  C-3 02 - - - -
B-4 01 - - - -  E-2 01 - - - -  C-3 03 - - - -
---    - - - -  E-2 01 - - - -  ---    - - - -
C-5 01 - - - -  G-2 01 - - - -  C-3 02 - - - -
---    - - - -  A-2 01 - - - -  ---    - - - -
C-4 01 - - - -  C-3 01 - - - -  C-3 03 - - - -
```

… and would sound like:

0:00 / 0:03

Though this example is musically dubious, the principle of height is shown: the height of the position is decorrelated from how many notes your tracks may contain, which makes re-usability greater.

Transpositions

The concept of re-usability is even more enhanced with the principle of transposition. In music, a transposition is simply an addition or subtraction of one or more semi-tones to notes.

> Transposition in practice: imagine you show someone how to sing a song. But their voice is much higher that your deep baritone voice. So you start singing higher so that it fits their voice better. You added a few semi-tones to the melody, and now the other person can sing it without hassle (but now you may be in trouble because you must sing higher. Ah well).

In AT, each position allows its tracks to use a different transposition setting. This is an awesome feature to save memory: a track can be use multiple times with a different transposition, which costs almost nothing. In the following example, our three previous positions will have various transpositions for the melody and the bass:

0:00 / 0:03

… I fully admit this is not very musical in the context, but the principle is applied. Can you guess much transposition is used? *(answer: +2 on the melody of position 0, +2 on the bass of position 1, +10 on the melody and -2 on the bass of position 2)*

Subsongs

The concept of subsongs appeared in AT2 and is pretty simple. Imagine you work on a game soundtrack. You will probably have several songs: a few long in-game tunes, start-game / game-over jingles, one high-score music, and so on. You could create as many songs as they are music. Or you could use subsongs.

A song can have up to 256 subsongs: each will be a music on its own, but instruments and expressions (arpeggios and pitches) will be shared. This is handy because:

- Using the same instruments/expressions from one subsong to another save memory.
- Your sound aesthetic will pass from a music to another naturally, making your soundtrack more cohesive.
- Technically speaking, it is even possible to split the music files in memory, loading only the parts you need, saving even more memory.

You don't *have* to use subsongs. It is only a handy possibility. There is always at least one subsong in a song, created by default.

Conclusion

With all this, I hope you have a better understanding of how to sequencing is done in AT. Other tutorials will show you how to compose a song properly!

> AT1/2/STarKos users may be shocked right now: patterns didn’t exist, since tracks were referred to directly, allowing to reuse the latter directly. Well, this has been simplified. Patterns allow you not to get lost with numbers, lost tracks and so on (and managing more than 3 tracks with all these numbers was really a chore). Don’t worry about optimization, this is also taken care of automatically by AT, but you can also make references by yourself (advanced concept presented here).

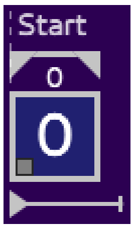
Creating a first melody

Let’s have some fun and create a first song! But first of all, I strongly encourage you to read the two previous tutorials, the one about the PSG and the one about the concept of sequencing.

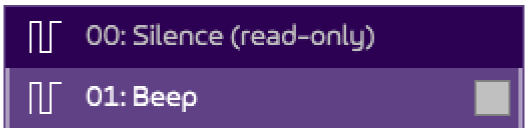
> Note that this tutorial relies on the default keyboard mapping, so if you’ve tweaked it, please don’t blame me if the shortcuts shown here are not the same as yours. If at any point you wonder how to do any keyboard action, don’t hesitate to go to File > Setup > Keyboard mapping. This is also a good way of learning of actions not explained in this manual yet handy to you.
>
> There may be some keys that are not working well depending on your OS and your keyboard layout. Remapping them will correct the problem!

The beginning

Arkos Tracker (**AT**) starts with an empty song. This is a good start. The software also kindly create one empty pattern, as you can see in the Linker (**LK**) at the top-right:



Also, in the instrument list (**IL**) on the left is one basic instrument called “Beep”, which we will use it to create our first melody. We’ll create our own a bit later.



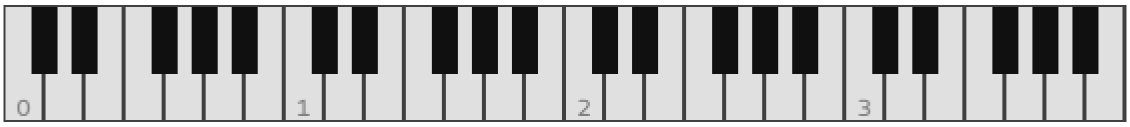
Let’s hear something!

Before laying our notes down, we want to make sure our *beep* sound can actually be heard. Maybe your sound card is faulty? Or my software buggy? There are three way for playing notes: you can use **computer keyboard**: AZERTY owners will pressed a to y, QWERTY owners: q to y. How fun! A small beep should be heard (if not, consult the File > Setup > Audio/MIDI interfaces). Notice how the more “right” you go on your keyboard, the higher the note is .

You can also use lower notes by pressing w to n for AZERTY owners, z to n for QWERTY owners. These are the sames notes, one octave below.

Want more notes? You can select the base octave either by changing the “octave” at the top of the screen, or by pressing numpad + to increase it, numpad - to decrease it. After changing the octave, play some notes to hear the change.

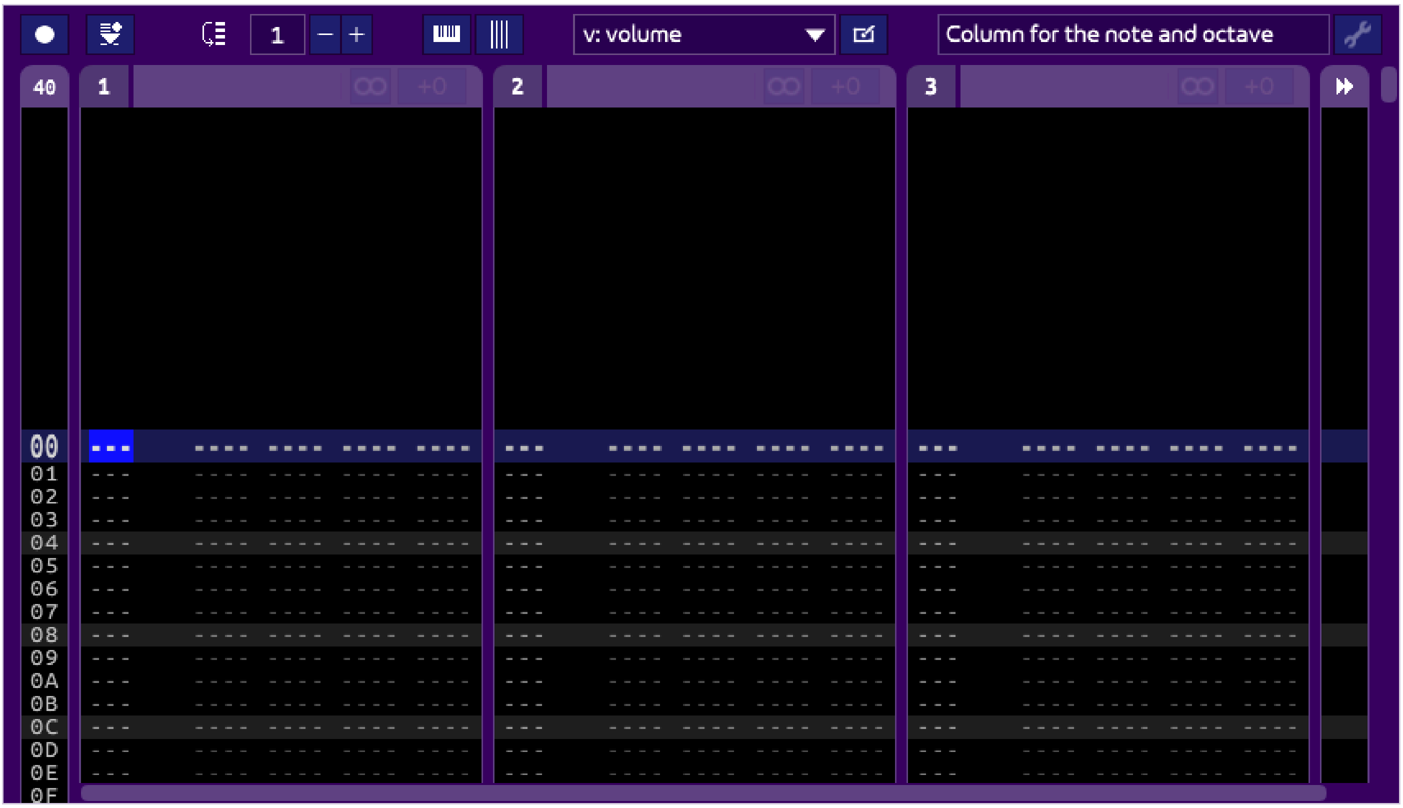
You can also click on the **piano at the bottom** to emit some notes:



Or if you have a **MIDI keyboard**, plug it and use it! You will probably have to set it up in the File > Setup > Audio/MIDI interfaces page, which is also detailed here.

Writing our first melody

Everything is ready to write our first (great) melody. Above the piano area and taking most of the space is the pattern viewer (PV). This is the main element you will be working with.



If not done already, give focus to it, either by clicking inside it, or by pressing F2 (remember this key!). Notice how pressing the QWERTY / AZERTY keys produce sounds. However, nothing is written yet.

What you see is called a **pattern**. It is simply a group of tracks. Three to be precise, because the default song targets an Amstrad CPC, which PSG can handle 3 channels. So far, all three tracks are empty. You should see your cursor in blue. Use the updownpage up page down keys to change the line where your cursor is. By using homeend, you can move to the first and last line of your pattern. The last line should be marked “3F”, or 63 in decimal.

By using rightleft, the cursor moves horizontally, from one column to the another. Even simpler, left-click on a location for the cursor to go to it!

You can clearly see your 3 tracks. Each is composed of many columns. What the hell is that? Don’t worry, it’s simple. Plus, a little help at the top-right of the pattern indicates what is under the cursor, so you’ll never get lost:

Column for the note and octave

Move your cursor at the top-left of your first track. It looks like the picture of the PV above.

We will write our first note. For this to be possible, the Record mode must be on. Either click on the Record button at the top of the PV ●, or press ctrl + space (or on Mac ctrl + r). Look at the Record button: it is bright red, so is the border of the PV.

Makes sure the octave is 3 (you've learned how to check and change it before).
Press a (AZERTY) or q (QWERTY). Yay! Our first note! You should see `C-4 01`. What does it mean? Simple: "C" is the note, using the US notation ("C" for "do", "C#" for "do#", etc. In order: `C, C#, D, D#, E, F, F#, G, G#, A, A#, B`).
"4" means the octave 4 is used. "01" means instrument 1, which is "Beep".

The other columns are reserved for effects, which we will use only later, so let's forget about them for now.

> The pattern viewer is explained thoroughly in its specific page. You can keep it in a corner of your browser as a reference.

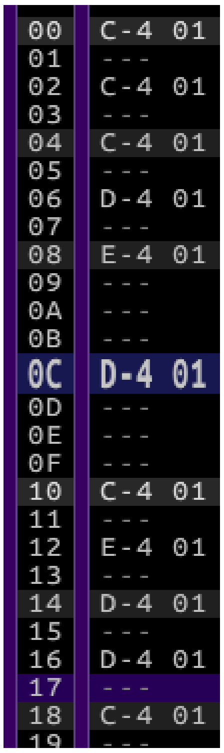
You have written one note of our masterpiece. Let's hear it again. But how? There are several ways.

You can use the icons at the top of the screen, such as ▶ to play the song from the start, ▼ to play the current pattern from the top, and ■ to stop playing.

Keyboard shortcuts exist, and I suggest you to use them. Here are only a few:

- space to start/stop playing the current pattern from the top (it also does more, which will be explained later).
- enter to play the note under the cursor and go to the next line. Very handy to play a few notes only!
- F9 to play the whole song from start.
- F11 to play the current pattern from the top.
- escape to stop playing.

One note does not make a song. Let's add more! Go to the top of the pattern, first track, and write the little song described below. Only three different notes are required (the first three notes of your top keyboard! And if you make mistakes, use del to erase, or ctrl + z to undo).



```
00  C-4 01
01  ---
02  C-4 01
03  ---
04  C-4 01
05  ---
06  D-4 01
07  ---
08  E-4 01
09  ---
0A  ---
0B  ---
0C  D-4 01
0D  ---
0E  ---
0F  ---
10  C-4 01
11  ---
12  E-4 01
13  ---
14  D-4 01
15  ---
16  D-4 01
17  ---
18  C-4 01
19  ---
```

This a rendition of the famous “Au clair de la lune / By the light of the moon”. No need to do something more complex for now. Don’t worry, we will spice it up quite soon.

Note: you may wonder why I chose to have one empty line between each note, where I could have saved some room by packing them all and (probably) decrease speed. One answer is that it allows to add more notes in between. So by having more space, you also get more resolution to put more notes. But the answer is mostly personal. Some musician like to have more space between notes “just in case”, or because they like having a quick-scrolling pattern. Anyway, just be assured that there is no wrong answer and, given that you can change the speed at any time (we’ll see how), you can have one pattern with packed data, and another with many “holes”.

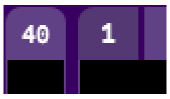
Pattern height

By now, if you entered the notes correctly, you should hear this:

0:00 / 0:07

This is great (yes it is!), but the pattern is too long: it would be nice if the melody could be played without the big blank that comes after it. There are different way on handling this, depending on your music. You can add new notes if you consider your melody should continue. Or you can shrink the pattern. This is what we are going to do.

At the top left of the PV is a number “40”. It is actually the length of the **position**.



Yes, I said position, and not pattern. That is because, the pattern itself has no height. The height is defined in the position. This is nice: you can use the same pattern several times with a possibly different height each time.

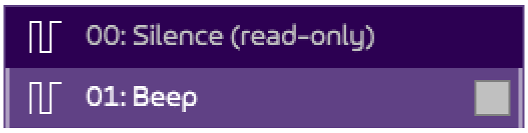
Click on the “40”: this opens a dialog for you to define a new height. Set it to 20 (hex), thus half the current height. Play the pattern again: it has now the “right” height, musically speaking.

Let’s continue on the next page!

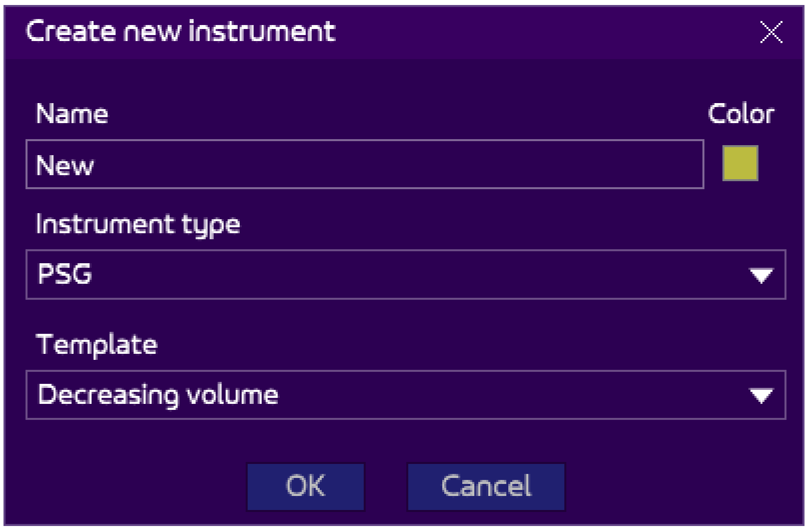
Creating our first instrument

This is the second part of the tutorial.

There is a melody, now we'll add a bass. First, we'll create another sound for this. On the left of the screen is the Instrument List. There is already the default sound, called Beep, which we used for our melody.

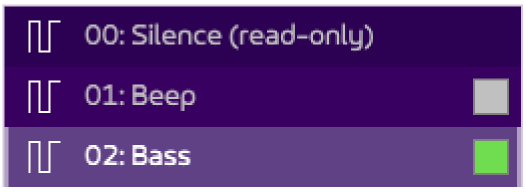


To create a new instrument, you can either click on the "plus" icon at the bottom of the list, or right-click on the list and select "Create at the bottom". Once clicked, a dialog opens:

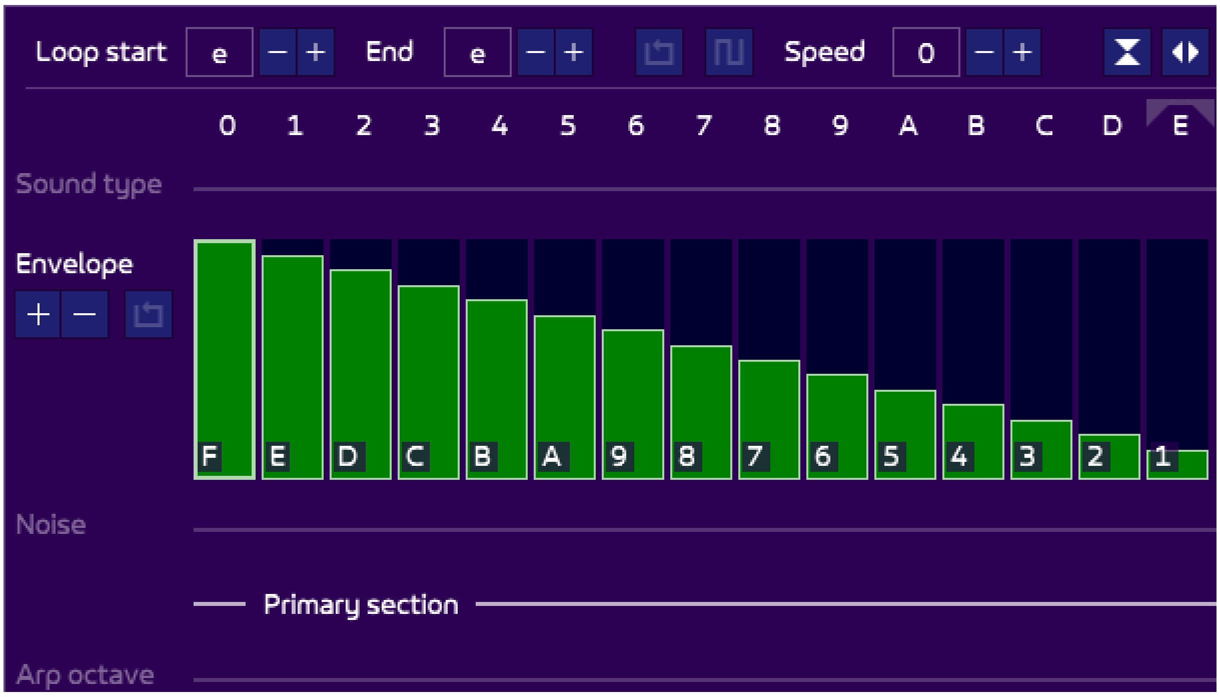


It allows you to name the new instrument (change it to "Bass"), and select a template for it: this can save you some time to have an already formed instrument which you can shape into another one. In order to learn how the thing works, select "PSG" as the instrument type, and "Decreasing volume" as the template: this will create a very short and simple sound, which we will improve. You can also select a nice color, which will show in the Pattern Viewer later.

Press OK. On the left, you can now see the bass instrument on slot 2:



Double-click on it to open it in the instrument editor (IE). It looks like this:


Loop start e − + End e − + Speed 0 − +

0 1 2 3 4 5 6 7 8 9 A B C D E

Sound type

Envelope

F E D C B A 9 8 7 6 5 4 3 2 1

Noise

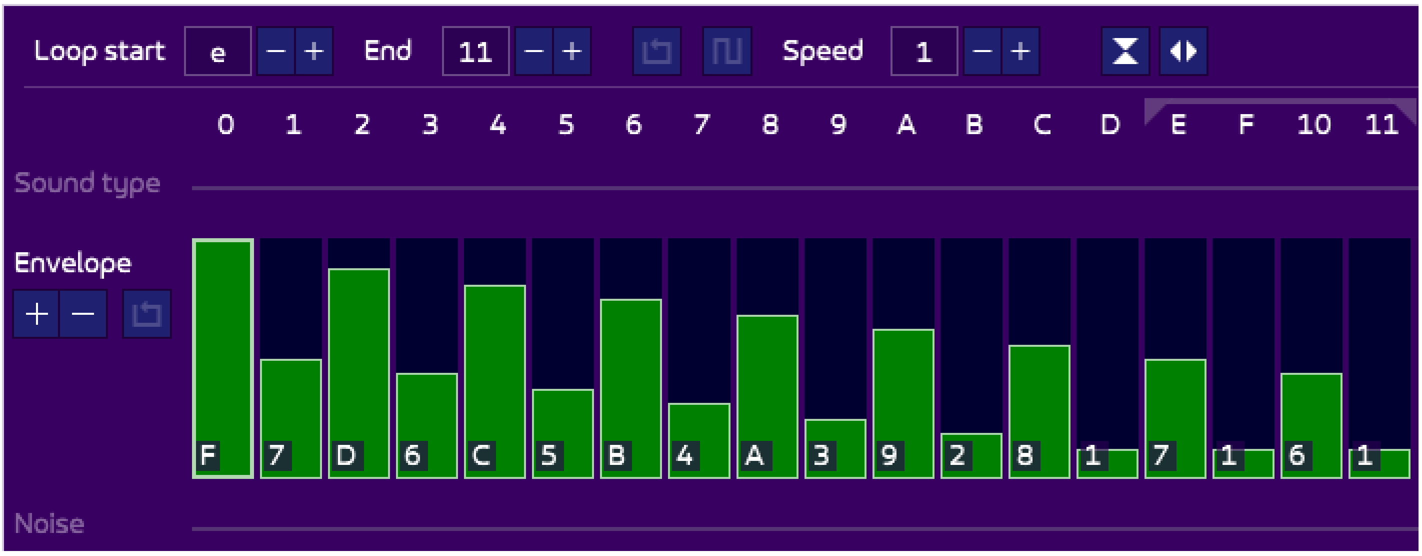
Primary section

Arp octave

Don't be afraid about all these bars. Press the usual qwerty keys to listen to the new sound. It is not a very exciting one.

0:00 / 0:02

There are several ways to edit the instrument, which are thoroughly explained in this page. We will take the less scary one: with the mouse. Let's draw an original curve. Point your mouse on the envelope curve, press ctrl and draw a nice curve. Try to make it look like mine:


Loop start e − + End 11 − + Speed 1 − +

0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11

Sound type

Envelope

F 7 D 6 C 5 B 4 A 3 9 2 8 1 7 1 6 1

Noise

Note: like in the PV, you can Undo anything, and press del to suppress the bar where the cursor is (left-click to move it, or use the cursor).

Note also two things:

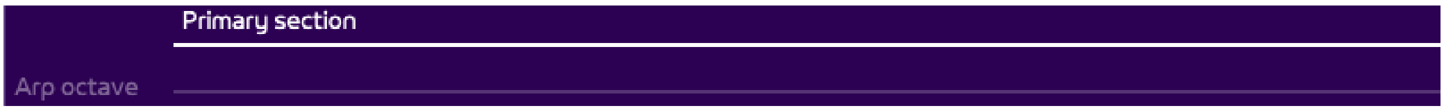
- The speed at the top-right is set to 1. The higher the speed, the slower the sound. This may sound illogical, but think of it this way: it is actually the delay between each column. Have fun and choose a speed that fits you.
- I created more columns than there was before, hence the "end" at the top-left set to 11 instead of F previously. You can create as many columns as you want. You can set the "end" to a lower value if your sound is too large, or delete the unwanted columns by locating the cursor on it (right-click on it) and pressing del.

You can use the lower octave to play it, as it is our bass after all (numeric pad - numeric pad + to change the octave, or change the octave at the top of the screen).

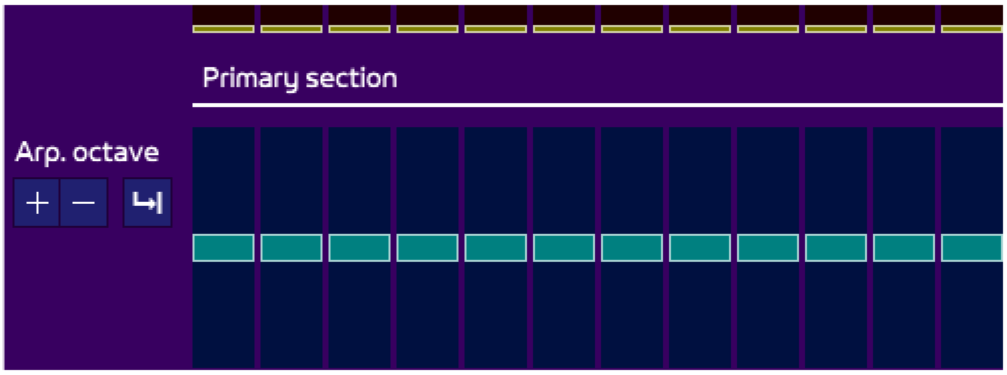
This should sound like this:

0:00 / 0:04

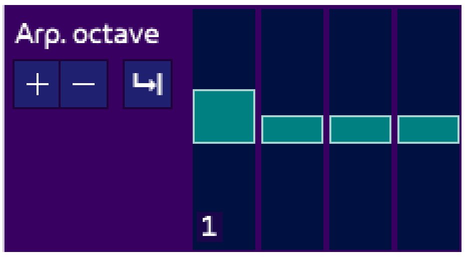
Rather original isn't it? But let's spice it up a bit. If needed, scroll down to reach the "Arp. octave" section in the **primary** section. Like most sections, it is collapsed (hidden) by default because there is no data in it: AT hides it to save space and make the interface less scary:



By hovering the mouse on the "Arp octave" title on the left, a small arrow appears. Click on it to show the section:



(By clicking on the arrow again, you can hide the row again). Right now there are no values. Move your cursor to the first column and change the value to octave 1:



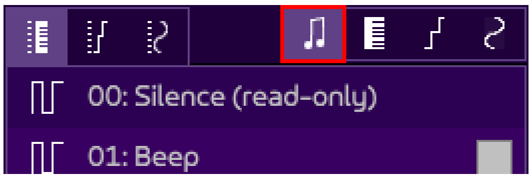
Play the sound a bit: now you should hear a nice attack with a higher octave.

0:00 / 0:04

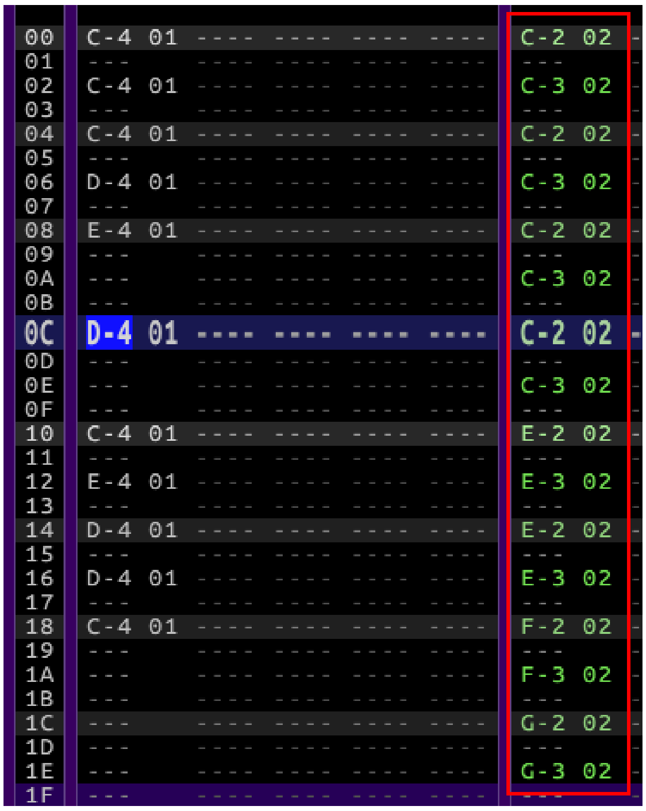
You can have some fun by adding more octave change in the other columns, but I'll leave this exercise to you. For now, let's consider this instrument finished. Let's use it!

Adding the bass in the pattern

Let's go back to the PV: you can do this either by pressing F2, or pressing on the graphical shortcut icon at the top-right of the IL:



Then move your cursor at the top of the pattern, on the second track (which should be empty). Make sure the "Bass" instrument is selected in the IL, and enter the following notes, highlighted in red:



```
00  C-4 01 ---- ---- ---- ----  C-2 02 -
01  ---    ---- ---- ---- ----  ---    -
02  C-4 01 ---- ---- ---- ----  C-3 02 -
03  ---    ---- ---- ---- ----  ---    -
04  C-4 01 ---- ---- ---- ----  C-2 02 -
05  ---    ---- ---- ---- ----  ---    -
06  D-4 01 ---- ---- ---- ----  C-3 02 -
07  ---    ---- ---- ---- ----  ---    -
08  E-4 01 ---- ---- ---- ----  C-2 02 -
09  ---    ---- ---- ---- ----  ---    -
0A  ---    ---- ---- ---- ----  C-3 02 -
0B  ---    ---- ---- ---- ----  ---    -
0C  D-4 01 ---- ---- ---- ----  C-2 02 -
0D  ---    ---- ---- ---- ----  ---    -
0E  ---    ---- ---- ---- ----  C-3 02 -
0F  ---    ---- ---- ---- ----  ---    -
10  C-4 01 ---- ---- ---- ----  E-2 02 -
11  ---    ---- ---- ---- ----  ---    -
12  E-4 01 ---- ---- ---- ----  E-3 02 -
13  ---    ---- ---- ---- ----  ---    -
14  D-4 01 ---- ---- ---- ----  E-2 02 -
15  ---    ---- ---- ---- ----  ---    -
16  D-4 01 ---- ---- ---- ----  E-3 02 -
17  ---    ---- ---- ---- ----  ---    -
18  C-4 01 ---- ---- ---- ----  F-2 02 -
19  ---    ---- ---- ---- ----  ---    -
1A  ---    ---- ---- ---- ----  F-3 02 -
1B  ---    ---- ---- ---- ----  ---    -
1C  ---    ---- ---- ---- ----  G-2 02 -
1D  ---    ---- ---- ---- ----  ---    -
1E  ---    ---- ---- ---- ----  G-3 02 -
1F  ---    ---- ---- ---- ----  ---    -
```

This simple bass line now makes the pattern sound like this:

0:00 / 0:03

So much fun! Let’s go on to the next part…

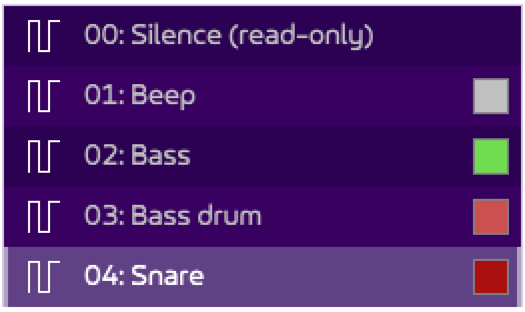
Drums, speed, and a second pattern

This is the third part of the tutorial.

We’ve got a melody and a bass line, let’s add drums! Drums are notoriously tricky to create, so we’ll use a shortcut and use the templates included to AT (there is another page dedicated to creating drums) *(we could also load any of the drums in the AT package by right-clicking on the Instrument List and select “Load instrument after”, but that’s not what we’re going to do here*).

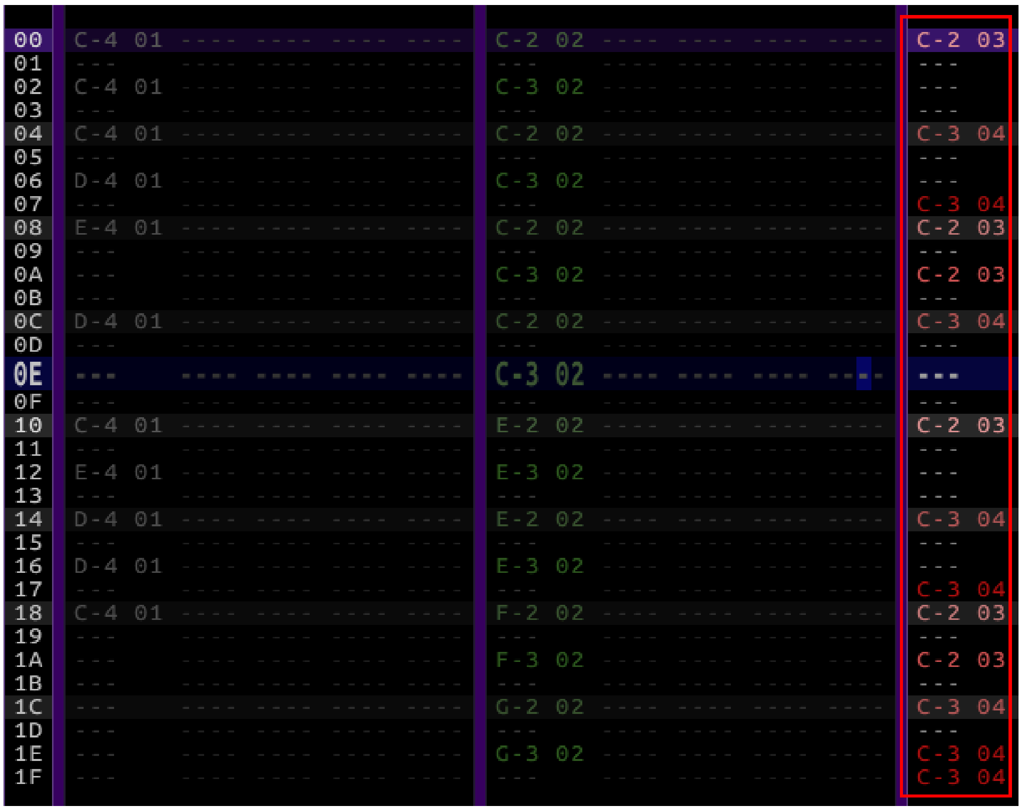
Create another sound in the Instrument List on the left: this time, call it “Bass drum” and select the like-named template. It is added in slot 3.

Do the same with a “Snare”, in slot 4. With these, we have this:



00: Silence (read-only)	
01: Beep	⬜
02: Bass	🟩
03: Bass drum	🟥
04: Snare	🟥

Open the PV and, on the third and empty track, add the following notes:



```
00  C-4 01 ---- ---- ---- ----   C-2 02 ---- ---- ---- ----   C-2 03
01  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
02  C-4 01 ---- ---- ---- ----   C-3 02 ---- ---- ---- ----   ---
03  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
04  C-4 01 ---- ---- ---- ----   C-2 02 ---- ---- ---- ----   C-3 04
05  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
06  D-4 01 ---- ---- ---- ----   C-3 02 ---- ---- ---- ----   ---
07  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   C-3 04
08  E-4 01 ---- ---- ---- ----   C-2 02 ---- ---- ---- ----   C-2 03
09  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
0A  ---    ---- ---- ---- ----   C-3 02 ---- ---- ---- ----   C-2 03
0B  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
0C  D-4 01 ---- ---- ---- ----   C-2 02 ---- ---- ---- ----   C-3 04
0D  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
0E  ---    ---- ---- ---- ----   C-3 02 ---- ---- ---- ----   ---
0F  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
10  C-4 01 ---- ---- ---- ----   E-2 02 ---- ---- ---- ----   C-2 03
11  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
12  E-4 01 ---- ---- ---- ----   E-3 02 ---- ---- ---- ----   ---
13  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
14  D-4 01 ---- ---- ---- ----   E-2 02 ---- ---- ---- ----   C-3 04
15  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
16  D-4 01 ---- ---- ---- ----   E-3 02 ---- ---- ---- ----   ---
17  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   C-3 04
18  C-4 01 ---- ---- ---- ----   F-2 02 ---- ---- ---- ----   C-2 03
19  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
1A  ---    ---- ---- ---- ----   F-3 02 ---- ---- ---- ----   C-2 03
1B  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
1C  ---    ---- ---- ---- ----   G-2 02 ---- ---- ---- ----   C-3 04
1D  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---
1E  ---    ---- ---- ---- ----   G-3 02 ---- ---- ---- ----   C-3 04
1F  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   C-3 04
```

Let's add new shortcuts to your arsenal:

- alt + up / alt + down to select the next/previous instrument.
- enter to select the nearest instrument from your cursor.

Let's listen to our "song":

0:00 / 0:03

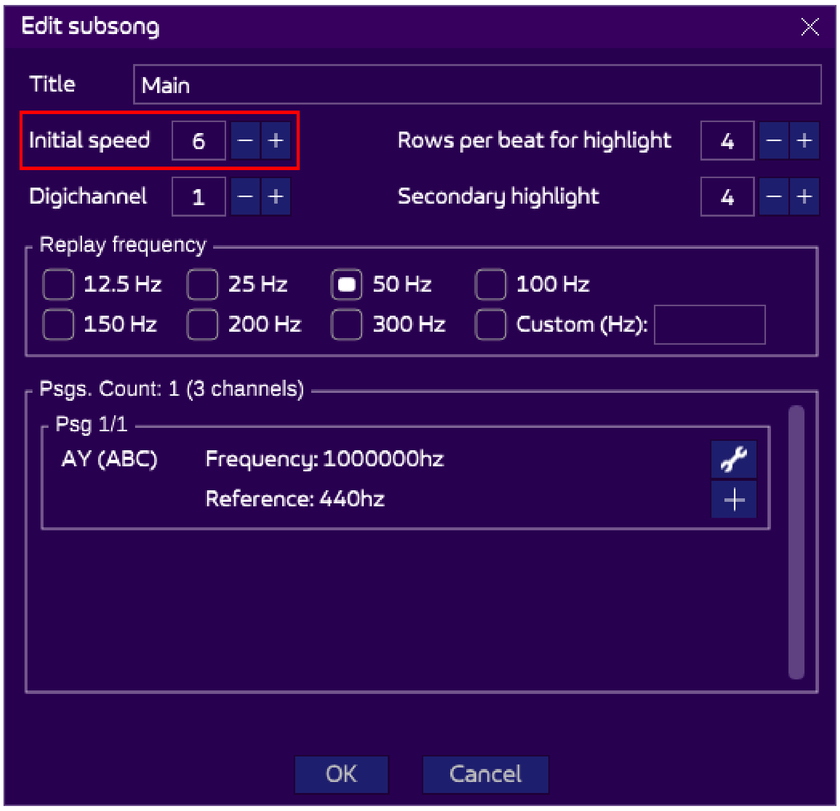
We've created our first pattern, yay! This could be the whole song, but we're going to extend it a bit very soon. But now, let's talk about…

Speed

Maybe your think this music has the right tempo. Or maybe not. I think it does, personally. However, here's a quick glimpse on how to change the speed. There are two ways.

Speed in song properties

This is the simplest way, but actually works in many cases. Each song has an *initial speed* parameter. Open the Edit > Subsongs properties > 0: Main:


Edit subsong

Title: Main

Initial speed: 6

Rows per beat for highlight: 4

Digichannel: 1

Secondary highlight: 4

Replay frequency: [ ] 12.5 Hz [ ] 25 Hz [✓ 50 Hz] [ ] 100 Hz [ ] 150 Hz [ ] 200 Hz [ ] 300 Hz [ ] Custom (Hz):

Psgs. Count: 1 (3 channels)

Psg 1/1

AY (ABC) Frequency: 1000000hz

Reference: 440hz

Highlighted in red is the speed that will be used at the beginning of your song. What means “6”? It means that each line of your track will last 6 *frames* before the next line is read. So, the lower speed, the fastest the song: a speed of 1 will have each line last one frame.

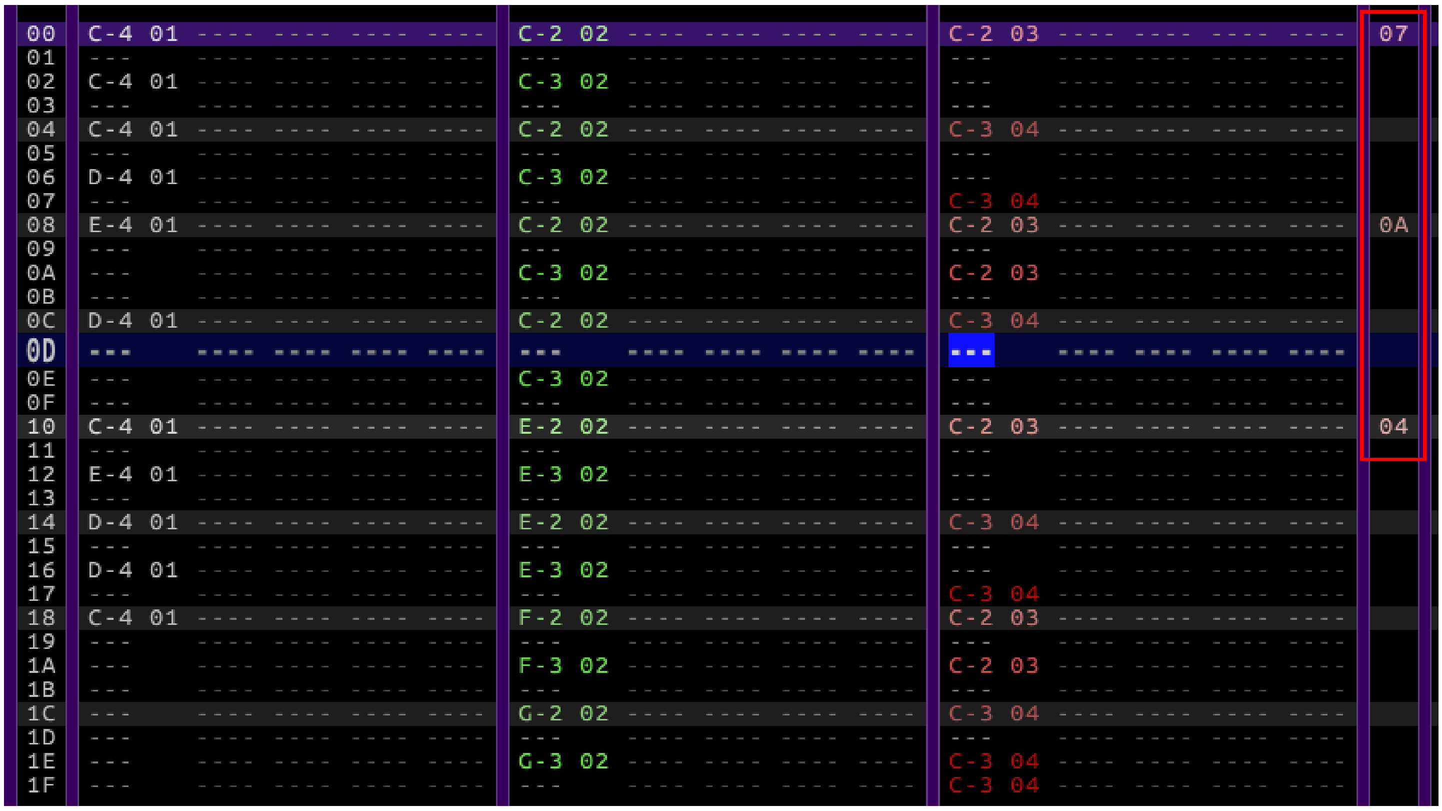
> The *frame* duration depends on the replay frequency which you can also set in this dialog. In most cases, you will use 50 Hz (because your hardware screen probably has a refresh rate of 50 Hz!). You can change this value, but be aware that this **has** technical consequences (especially if you raise it!), so ask advice from your coder first to make sure your song will actually work with their code.

Changing the initial speed works in many cases, but what if you want to change it during the course of the song? Well, a second technique is necessary.

Speed in the tracks

You can change the song speed at any time by writing it in the tracks. Specifically, in the *speed tracks*. At the right of the music tracks are two smaller tracks, namely *speed* and *event* tracks. These are described in details here.

You can type a number from `01` (faaast) to FF (very slooooow), `06` being the default mid-tempo. As an example, writing the following tempos…



```
00  C-4 01 ---- ---- ---- ----  C-2 02 ---- ---- ---- ----  C-2 03 ---- ---- ---- ----  07
01  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
02  C-4 01 ---- ---- ---- ----  C-3 02 ---- ---- ---- ----  ---    ---- ---- ---- ----
03  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
04  C-4 01 ---- ---- ---- ----  C-2 02 ---- ---- ---- ----  C-3 04 ---- ---- ---- ----
05  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
06  D-4 01 ---- ---- ---- ----  C-3 02 ---- ---- ---- ----  ---    ---- ---- ---- ----
07  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  C-3 04 ---- ---- ---- ----
08  E-4 01 ---- ---- ---- ----  C-2 02 ---- ---- ---- ----  C-2 03 ---- ---- ---- ----  0A
09  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
0A  ---    ---- ---- ---- ----  C-3 02 ---- ---- ---- ----  C-2 03 ---- ---- ---- ----
0B  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
0C  D-4 01 ---- ---- ---- ----  C-2 02 ---- ---- ---- ----  C-3 04 ---- ---- ---- ----
0D  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
0E  ---    ---- ---- ---- ----  C-3 02 ---- ---- ---- ----  ---    ---- ---- ---- ----
0F  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
10  C-4 01 ---- ---- ---- ----  E-2 02 ---- ---- ---- ----  C-2 03 ---- ---- ---- ----  04
11  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
12  E-4 01 ---- ---- ---- ----  E-3 02 ---- ---- ---- ----  ---    ---- ---- ---- ----
13  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
14  D-4 01 ---- ---- ---- ----  E-2 02 ---- ---- ---- ----  C-3 04 ---- ---- ---- ----
15  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
16  D-4 01 ---- ---- ---- ----  E-3 02 ---- ---- ---- ----  ---    ---- ---- ---- ----
17  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  C-3 04 ---- ---- ---- ----
18  C-4 01 ---- ---- ---- ----  F-2 02 ---- ---- ---- ----  C-2 03 ---- ---- ---- ----
19  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
1A  ---    ---- ---- ---- ----  F-3 02 ---- ---- ---- ----  C-2 03 ---- ---- ---- ----
1B  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
1C  ---    ---- ---- ---- ----  G-2 02 ---- ---- ---- ----  C-3 04 ---- ---- ---- ----
1D  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
1E  ---    ---- ---- ---- ----  G-3 02 ---- ---- ---- ----  C-3 04 ---- ---- ---- ----
1F  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  C-3 04 ---- ---- ---- ----
```

… will sound like this:

0:00 / 0:04

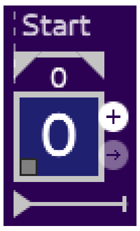
The first 8 lines (`00` to `07`) play at a speed of `07` (medium tempo). Then the lines from `08` to `0F` play even slower at `0A`. The remaining lines play at a speed of `04`, which is pretty fast. Funny, no?

This was only a (dubious sounding) example on how to use speed. You can remove these speed values, or add one at the top with a value of your choice.

Now let's continue this great song!

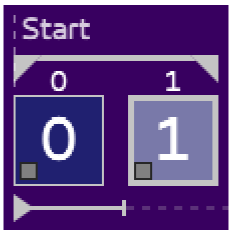
A second pattern

We are going to clone the position: in the linker a the top, hover your mouse above the first position: two small icons are going to appear:



The "+" icon is for creating a whole new empty pattern, the arrow is for duplicating. But what I want is cloning. Right-click on the position and select "clone". The difference? *Cloning* will create another position with a new pattern, which content is the same. It's like cloning a sheep: if you paint the second sheep red, the first one is still white. *Duplicating* would create another position with the *same* pattern: if you modify it, both position 0 and 1 will be impacted. That's nice, but not what I want here.

Clone the first position. Yay, a position 1 with a pattern 1 is now present:

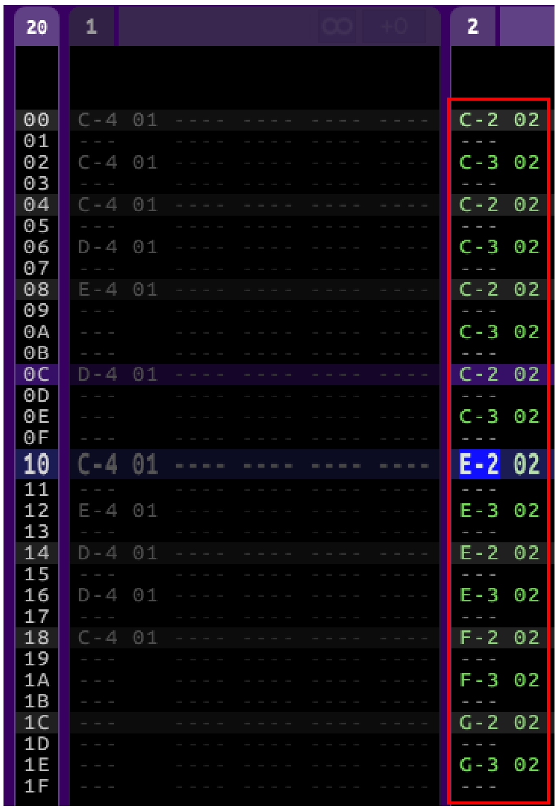


The second position is selected and its content can be seen in the Pattern Viewer. Play the pattern with space. The content is the same as before!

Move your cursor to the bass track (in the middle) and select the bass instrument (with any shortcut I gave you above). We're going first to clear it (make sure the Record mode is on):

- ctrl + t to select the whole track.
- ctrl + x to cut its content.

Now enter these notes instead (only the second half is modified):

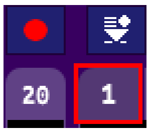


```
20  1                                 2
00  C-4 01 ---- ---- ---- ----   C-2 02
01  ---    ---- ---- ---- ----   ---
02  C-4 01 ---- ---- ---- ----   C-3 02
03  ---    ---- ---- ---- ----   ---
04  C-4 01 ---- ---- ---- ----   C-2 02
05  ---    ---- ---- ---- ----   ---
06  D-4 01 ---- ---- ---- ----   C-3 02
07  ---    ---- ---- ---- ----   ---
08  E-4 01 ---- ---- ---- ----   C-2 02
09  ---    ---- ---- ---- ----   ---
0A  ---    ---- ---- ---- ----   C-3 02
0B  ---    ---- ---- ---- ----   ---
0C  D-4 01 ---- ---- ---- ----   C-2 02
0D  ---    ---- ---- ---- ----   ---
0E  ---    ---- ---- ---- ----   C-3 02
0F  ---    ---- ---- ---- ----   ---
10  C-4 01 ---- ---- ---- ----   E-2 02
11  ---    ---- ---- ---- ----   ---
12  E-4 01 ---- ---- ---- ----   E-3 02
13  ---    ---- ---- ---- ----   ---
14  D-4 01 ---- ---- ---- ----   E-2 02
15  ---    ---- ---- ---- ----   ---
16  D-4 01 ---- ---- ---- ----   E-3 02
17  ---    ---- ---- ---- ----   ---
18  C-4 01 ---- ---- ---- ----   F-2 02
19  ---    ---- ---- ---- ----   ---
1A  ---    ---- ---- ---- ----   F-3 02
1B  ---    ---- ---- ---- ----   ---
1C  ---    ---- ---- ---- ----   G-2 02
1D  ---    ---- ---- ---- ----   ---
1E  ---    ---- ---- ---- ----   G-3 02
1F  ---    ---- ---- ---- ----   ---
```

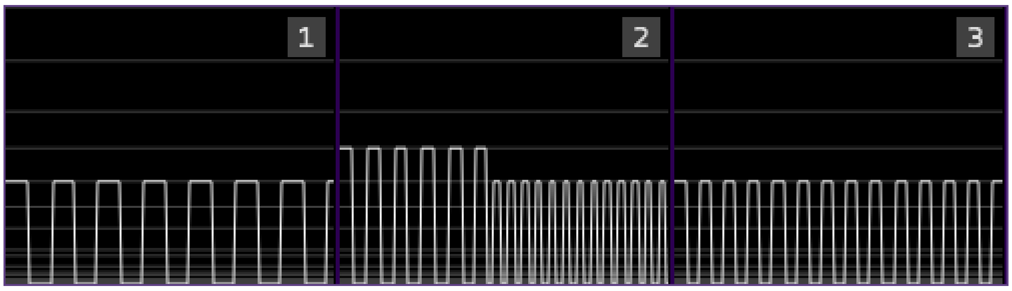
Solo/mute

To hear the difference, let's focus on one track only: there are two ways to mute a track, or isolate it:

left-click on the channel number at the top of each track.



left-click on the channel EQ in the Meters at the top-left of the screen.:



A muted channel appears greyed out, both in the PV and in the Meters.

To solo a channel, it's the same, but with a right-click!

You can also use the following shortcuts in the PV:

- ctrl-numpad 0 to mute the channel where your cursor is.
- ctrl-numpad 1 to solo the channel where your cursor is.

By soloing the bass track, you can hear the following:

0:00 / 0:03

And the whole song sounds like this:

0:00 / 0:07

What we've just done is duplicating the whole first position, and modify the bass on the newly created position. The other two tracks are still the same, which is fine to our music, and which you'll probably do a lot in your future songs.

> Some of you may be concerned that these duplications will use more memory than needed (and that the low-level yet cumbersome track management of STarKos/AT1/AT2 is more optimized). Don’t worry about this! AT3 will remove all same-sounding tracks!

Let’s move on to the next part of this tutorial…

Another melody and effects

This is the fourth part of our tutorial.

So far we've got two positions with two patterns. We're now going to create a new melody and add the first effects.

I want to clone the first position and put it at the end in order to modify it. Let me show two ways of doing it:

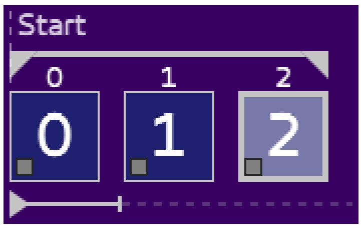
Manually

- We could create a new empty one: right-click on the last position and select "New", or press ctrl-n create the new pattern.
- Then you would have to copy/paste the content of the position 0 by using, in the PV ctrl-a to select the whole pattern, then copy/paste it in the new empty pattern.

Cloning and moving

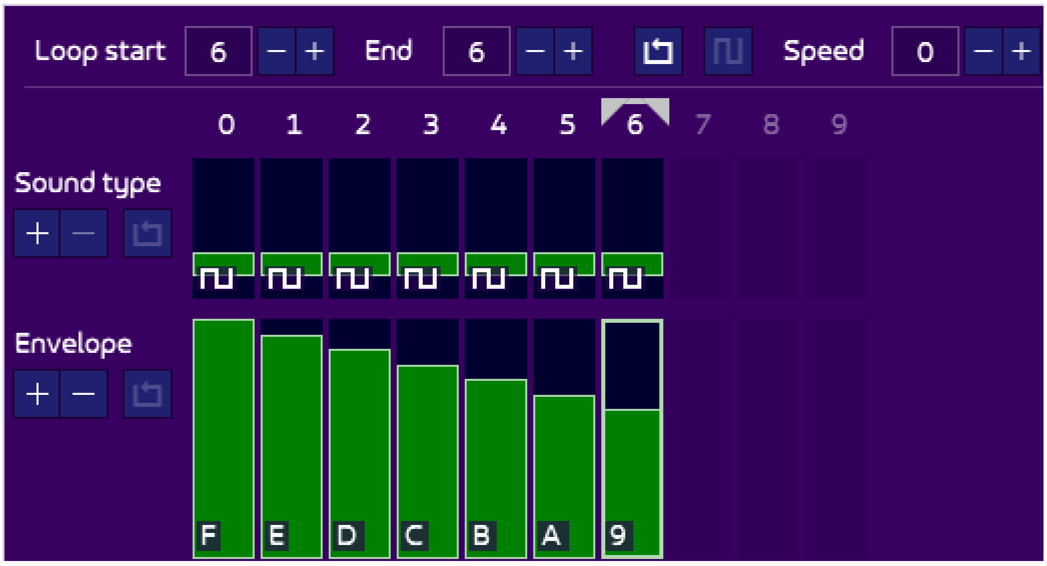
Or you could right-click on the position 0 in the Linker and select "Clone" (or use shift-insert). In position 1, a pattern 2 has been created. Drag'n'drop it to the end of the song.

In the end, your song should look like that:



A melody instrument

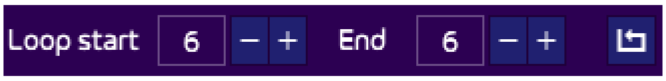
We're going to create a new instrument called Melody. You can use any template you want, but make it sounds a bit like that:



(The sound type section is shown, but it may be hidden on your screen). On this screenshot, you can see that there is a loop on the last bar: click on the numbered-"6" volume bar to move your cursor here, and press:

- ctrl-i to set the start loop.
- ctrl-o to set the end loop.
- ctrl-p to toggle the loop to on.

You could also use the sliders at the top to set the values manually:



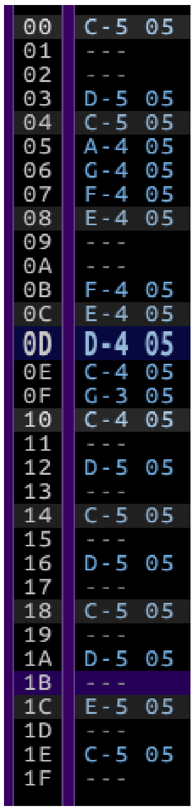
If you play the sound, it should have a nice attack sound that loops endlessly:

0:00 / 0:03

Writing the melody

Go back to the PV, and make sure that you are on the position 2, thus editing the pattern 2. Clear the notes of the first channel (the melody), as we're going to add a new one.

Select the Melody instrument (slot 5), and write these notes:



```
00  C-5 05
01  ---
02  ---
03  D-5 05
04  C-5 05
05  A-4 05
06  G-4 05
07  F-4 05
08  E-4 05
09  ---
0A  ---
0B  F-4 05
0C  E-4 05
0D  D-4 05
0E  C-4 05
0F  G-3 05
10  C-4 05
11  ---
12  D-5 05
13  ---
14  C-5 05
15  ---
16  D-5 05
17  ---
18  C-5 05
19  ---
1A  D-5 05
1B  ---
1C  E-5 05
1D  ---
1E  C-5 05
1F  ---
```

It should sound like this:

0:00 / 0:03

Great! But we'll improve it by adding effects.

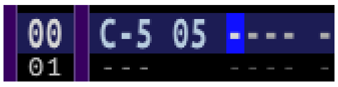
Volume effects

Now enters the four empty columns you can see in every track. In AT, you can use these columns to add effects, four maximum at the same time (which is more than enough). Effects are exactly what you think they are. If you play the guitar, you know what a vibrato is, or how you can make a slide from one note to another. Or if you sing, you know how soft or loud you can get. Such effects increase expressiveness. This is the same with AT effects.

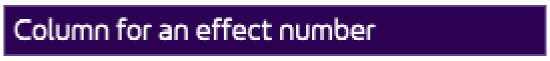
All the effects are described here, but in this tutorial I'm going to take it step by step – you can delve on the effect page later.

The first thing we're going to add is volume effects. This add dynamics to the sound. Volume can go from 0 (mute) to F (full volume), and the effect is represented by the letter "v" (each effect has its own assigned letter, which you can customize if wanted).

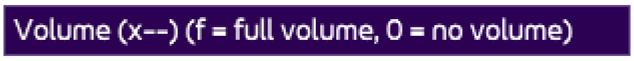
Go to the melody track, to the first effect column (which is empty), first line:


```
00  C-5 05 ---- -
01  ---    ---- -
```

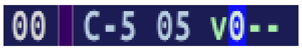
Notice that the Information View (**IV**) at the top-right hints at what is under the cursor, so that you don't get lost:


Column for an effect number

Put Record on, and type a "v". Go up to get back to the effect you just typed. The IV shows:


Volume (x--) (f = full volume, 0 = no volume)

This helps if you don't know how an effect works: this volume effect needs only one digit (hence `x--`), which you can see in the track – a default value of 0 is written along with the effect:

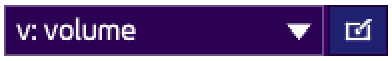

```
00  C-5 05 v0--
```

As we will see later, some effects use 2 or 3 digits. One thing you might have noticed is that your melody is not heard anymore. This is because the track now has a volume of 0, hence muted!

Replace the 0 with a F. Now the volume is full.

Note: the volume effect is not an amplifier, but actually an *attenuat*or. **F** means that the instrument volume is used at its full. **E** means that the instrument volume is decreased of 1. **D** of 2, **C** of 3 and so on.

We're going to make the melody more dynamic by adding volume at every note. We could go on and write "v" on every line, then the value, but this would be boring. At the top of the PV, you can see a drop-down:

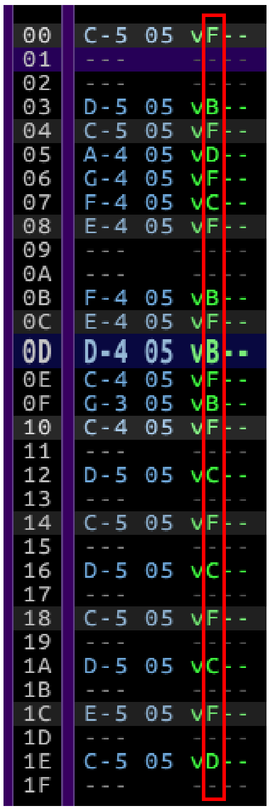


By clicking on it, you can see all the effects available in AT. For now, keep "volume" selected. This means that whenever you enter an effect value (such as the F written earlier), the drop-down effect will automatically be written. There are two advantages:

- You save some time because you don't have to type it.
- You don't have to remember the bloody effect letter!

> *Note:* be aware that by default, as a security, the drop-down effect will not overwrite an effect letter that would be present. To overwrite it in any case, toggle the icon at the right of the drop-down, which switches from to .

Now enter the following effect values:



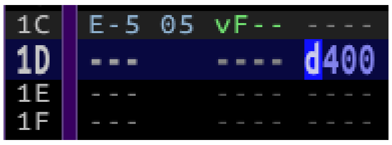
```
00  C-5 05 vF--
01  ---    ----
02  ---    ----
03  D-5 05 vB--
04  C-5 05 vF--
05  A-4 05 vD--
06  G-4 05 vF--
07  F-4 05 vC--
08  E-4 05 vF--
09  ---    ----
0A  ---    ----
0B  F-4 05 vB--
0C  E-4 05 vF--
0D  D-4 05 vB--
0E  C-4 05 vF--
0F  G-3 05 vB--
10  C-4 05 vF--
11  ---    ----
12  D-5 05 vC--
13  ---    ----
14  C-5 05 vF--
15  ---    ----
16  D-5 05 vC--
17  ---    ----
18  C-5 05 vF--
19  ---    ----
1A  D-5 05 vC--
1B  ---    ----
1C  E-5 05 vF--
1D  ---    ----
1E  C-5 05 vD--
1F  ---    ----
```

The melody sounds much more dynamic:

0:00 / 0:03

Pitch effect

Let me show another example with a *pitch* effect (which changes the frequency of the note). I'll remove the last note, and add a *pitch down* effect ("d"):


```
1C E-5 05 vF-- ----
1D ---    ---- d400
1E ---    ---- ----
1F ---    ---- ----
```

Two things to notice:

- I wrote the effect on the second column, but I could have put it on any other column. Do as you like! Personally, I like to put volume effects on the first column, and pitch effect on the second, for clarity.
- This effect requires 3 digits, contrary to the volume effect that required only one.

It sounds like this:

0:00 / 0:03

The “400” value may be obscure, but for now, simply accept that the bigger the value, the faster the pitch. Also note that even though I only wrote `d400` once, the pitch effect continues on the following lines. Pitch effects are *trailing effects*: they continue:

- as long as no other note is found.
- no other effect (of the same category (pitch, volume etc.)) is found.

You could also write `d000` to stop the effect (meaning, a pitch down that *doesn’t* pitch).

Note that the volume effect does not reset automatically. If you use a volume at 5, all the following notes will play at such low volume. Two possibilities to reset it:

- Write a vF on the next note.
- Use the *reset* effect (“r”) on the next note. The reset effect, as its name implies, stops the effects, puts the volume at its maximum, etc.

Conclusion

That’s it for this tutorial! I hope you enjoyed it and have a better understanding of AT3. Feel free to continue fiddling with this song, or smash the package songs to bits to understand the many musicians’ tricks. Many things are possible on AY, and many have still to be discovered!

About tempo and speed

Trackers have for decades used their own special way of managing tempo, that is, how fast your music plays. AT does not reinvent the wheel on this aspect, so what you may have learned on previous trackers still applies.

There is no tempo

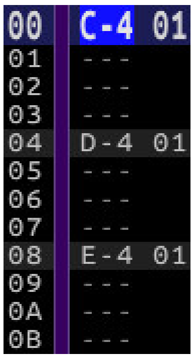
Indeed, old-school trackers don't really manage tempo, most of the time. The truth lies in technical aspects. In every old-school computer, 8 or even 16 bits, the easiest, and perhaps more reliable way of having an accurate notion of time passing by is… the screen refresh. Every 50 Hz (or 60 Hz in the US), the screen is refreshed and coders can synchronize on this event to produce smooth animations. This is also the perfect moment to play music!

But then it limits to playing a frame (or **tick**) of music every 50 Hz. 1 / 50 Hz = 20 ms. This might seems fast – and it is – but it doesn't allow using accurate tempo. You may get 125 **bpm** (Beat per minute), but not 126 or 127, because you don't have the required accuracy (more on that below).

There is speed

"**Speed**" is a term coined by old-school trackers – I don't know which one – and it's also the term used in AT, though, as we will see, it is actually a dubious term.

Imagine you play a note every tick, thus every 20 ms… You will actually have a tempo of 1500 bpm. Not very musical! Trackers solve this by having **lines** of notes, and by grouping them into **beats**:


```
00  C-4 01
01  ---
02  ---
03  ---
04  D-4 01
05  ---
06  ---
07  ---
08  E-4 01
09  ---
0A  ---
0B  ---
```

Typically, a group of 4 lines (here, 0 to 3, 4 to 7 and so on) forms a beat. Why 4? Because it is good compromise between readability (notes are grouped together and not spread across several screens) and accuracy (you can write quarter notes, most song won't need eighth).

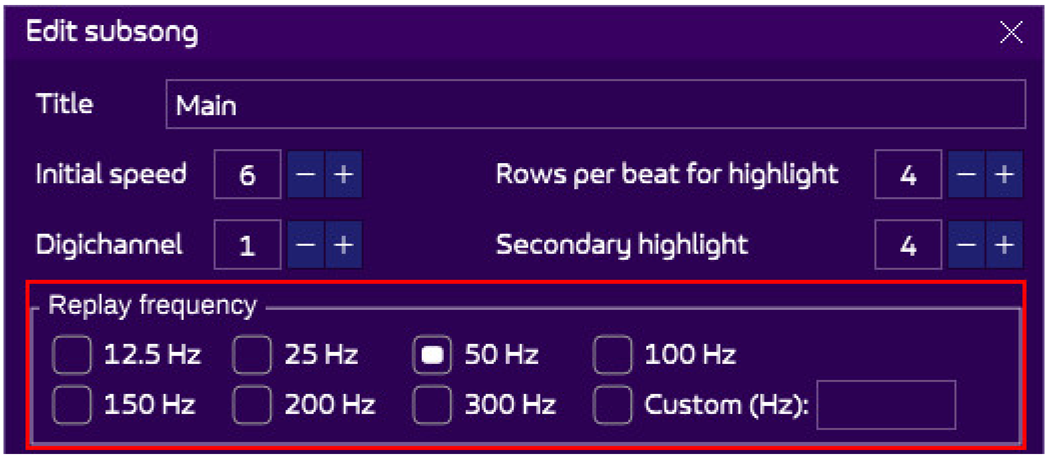
The speed, each unit a multiple of 20 ms, indicates the duration the player will wait before going to the next line. So with a speed of 1, going from line 0 to 1 will take 20 ms (very fast). A speed of 2 will double this amount, 40 ms (still very fast). A speed of 6 produces an average tempo: 125 bpm (calculation below), which is why it is default is many trackers, including AT.

So here is the "lie" with the "speed" term: the higher the speed, the lower the tempo! Which is why some trackers use the term *delay*. I don't, because I consider "speed" a speed, regardless of its scale (plus, *delay* is the name of a well known effect in music, i.e. echo).

More tempos?

You'll argue that some trackers, even old-school, allow to choose accurate tempos (126, 127, etc.). It may be possible on hardware with precise timers, like Amiga or Atari ST. Instead of waiting for the 50 Hz of the screen fly-back, they can use programmable timers to trigger the music replay at will. Since AT primary targets 8-bit computer without such capabilities, the choice of using of less-accurate *speed* has been made.

However, AT allows you to get such accurate tempos: you just have to use another replay frequency in the subsong properties:


Edit subsong

Title: Main

Initial speed: 6

Rows per beat for highlight: 4

Digichannel: 1

Secondary highlight: 4

Replay frequency: 12.5 Hz, 25 Hz, [✓ 50 Hz], 100 Hz, 150 Hz, 200 Hz, 300 Hz, Custom (Hz):

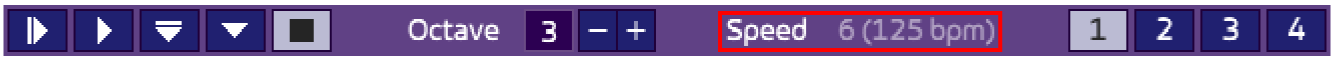
Instead of 50 Hz, use 59, 69, or any other value! **But I'm warning you:** unless you're sure such frequency can be used, do not!

However, even if the hardware may not be able to produce such accurate timer, it can often be done by software means (the code can wait for as long as needed and play another frame of music – but it is seldomly used in practice – it can be cumbersome to have animations or effects besides). Divisions of 50 Hz are easy to handle however (25, 12.5) and often used, especially in games. Instead of calling the player every frame, play it every 2 or 3 times. It saves CPU and may not even be noticeable, depending on the song (listen to the fantastic Stormlord soundtrack by Dave Rogers!)

On a CPC, we are somewhat lucky to have 300 Hz interruptions (6 interruptions during a 50 Hz screen fly-back), so it's *technically* easy to play such frequencies (though heavy on CPU!), as well as multiplications of 50 Hz: 200, 150, 100 Hz. Other 8-bit may not have that facility, so once again, make sure the song can be played on your hardware (and by a coder!).

Displayed BPMs

On top of the screen, you can see the speed and a BPM:



In AT, you cannot directly edit the speed via BPM, but you can set the speed itself, either via the speed track, or the *initial speed* of the subsong properties (see the screenshot above). Once changed, the BPM will update accordingly.

The calculation of the BPM

The BPM is calculated via different parameters:

- The speed
- The replay frequency (50 Hz, etc.).
- The rows per beat

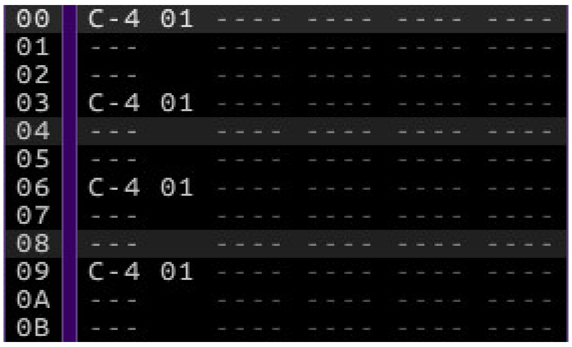
The formula is as is:

```
bpm = 60 / ((speed * rows per beat) * 1 / (replay frequency in Hz))
```

We've talked about the two first items, but what is the last one? In AT, it is only a parameter which you can find once again in the subsong properties: *the rows per beat for highlight*.

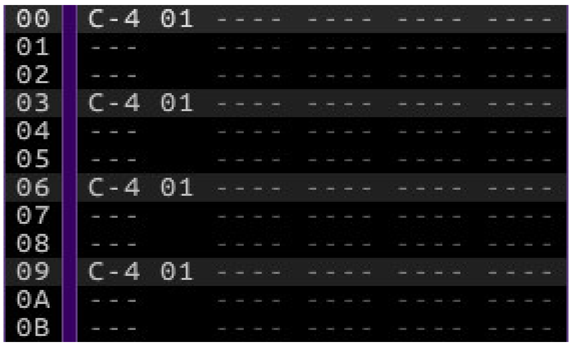
It is *only* for display, and changing it will *not* change the speed, but *will* change the displayed tempo. Huh?

Imagine you're composing a music in 3/4, so a triplet in a beat. Visually, you will not want this:



```
00  C-4 01 ---- ---- ---- ----
01  ---    ---- ---- ---- ----
02  ---    ---- ---- ---- ----
03  C-4 01 ---- ---- ---- ----
04  ---    ---- ---- ---- ----
05  ---    ---- ---- ---- ----
06  C-4 01 ---- ---- ---- ----
07  ---    ---- ---- ---- ----
08  ---    ---- ---- ---- ----
09  C-4 01 ---- ---- ---- ----
0A  ---    ---- ---- ---- ----
0B  ---    ---- ---- ---- ----
```

Notice how the highlights of the beats are on lines 4 and 8. Though it has no consequence on the music, this is not convenient for the composer. Change the *rows per beat* to 3:



```
00  C-4 01 ---- ---- ---- ----
01  ---    ---- ---- ---- ----
02  ---    ---- ---- ---- ----
03  C-4 01 ---- ---- ---- ----
04  ---    ---- ---- ---- ----
05  ---    ---- ---- ---- ----
06  C-4 01 ---- ---- ---- ----
07  ---    ---- ---- ---- ----
08  ---    ---- ---- ---- ----
09  C-4 01 ---- ---- ---- ----
0A  ---    ---- ---- ---- ----
0B  ---    ---- ---- ---- ----
```

Better looking! As a musician, this helps you see the triplets. But what's more, AT can now calculate the BPM now that it knows there are only 3 lines per beat.

More notes?

Let's go back to 4/4 as in our example at the top of this page. There are four notes per beat. Now this might be enough, but what if you want, during all the song, or only a few bars, have more notes because you have a more intricate melody?

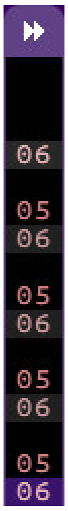
Simply double both the speed and the rows per highlight! The displayed tempo will be the same, yet you will have twice more notes scrolling twice faster.

This technique has some limits, which may or may not apply to you:

- It may be more exhausting for the eyes, since the scrolling is faster!
- Positions can only be 128-line height, so you may need to split your pattern in two if you were already using large positions.
- AT only supports only value of notes per beats per subsong. If only parts must have "more notes", the notes per beat will not change. This has no consequence on the song, only on the visuals, and the displayed tempo will not be accurate (twice higher than it should).

More accurate tempos

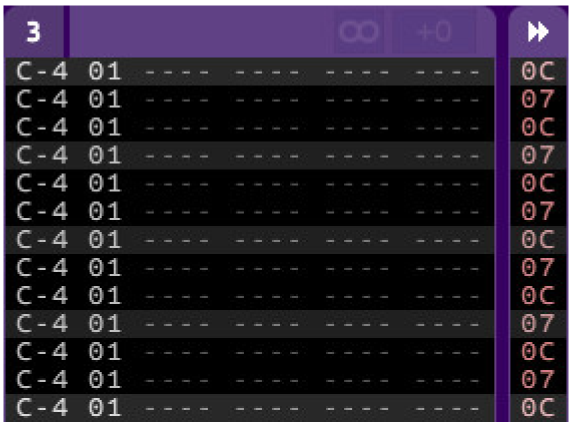
What if your tempo is too slow with a speed of 6, yet too fast with a speed of 5? If you are stuck with a 50 Hz player (which you probably are), a well know trick is to change the speed quickly. How much depends on you only. AT will not show an “average” bpm, so you will have to *feel* it. You could alternate between 6 and 5 on every line. Or something more subtle:



```
06

05
06

05
06

05
06

05
06
```

Groove and shuffle

An alternative use of such technique is to switch two very different speeds…



```
3
C-4 01 ---- ---- ---- ----   0C
C-4 01 ---- ---- ---- ----   07
C-4 01 ---- ---- ---- ----   0C
C-4 01 ---- ---- ---- ----   07
C-4 01 ---- ---- ---- ----   0C
C-4 01 ---- ---- ---- ----   07
C-4 01 ---- ---- ---- ----   0C
C-4 01 ---- ---- ---- ----   07
C-4 01 ---- ---- ---- ----   0C
C-4 01 ---- ---- ---- ----   07
C-4 01 ---- ---- ---- ----   0C
C-4 01 ---- ---- ---- ----   07
C-4 01 ---- ---- ---- ----   0C
```

… to create what musicians know as *shuffle*:

0:00 / 0:02

Nice, eh? The combinations are multiple and the result varied, so it’s up to you to make tests to check what sounds the best!

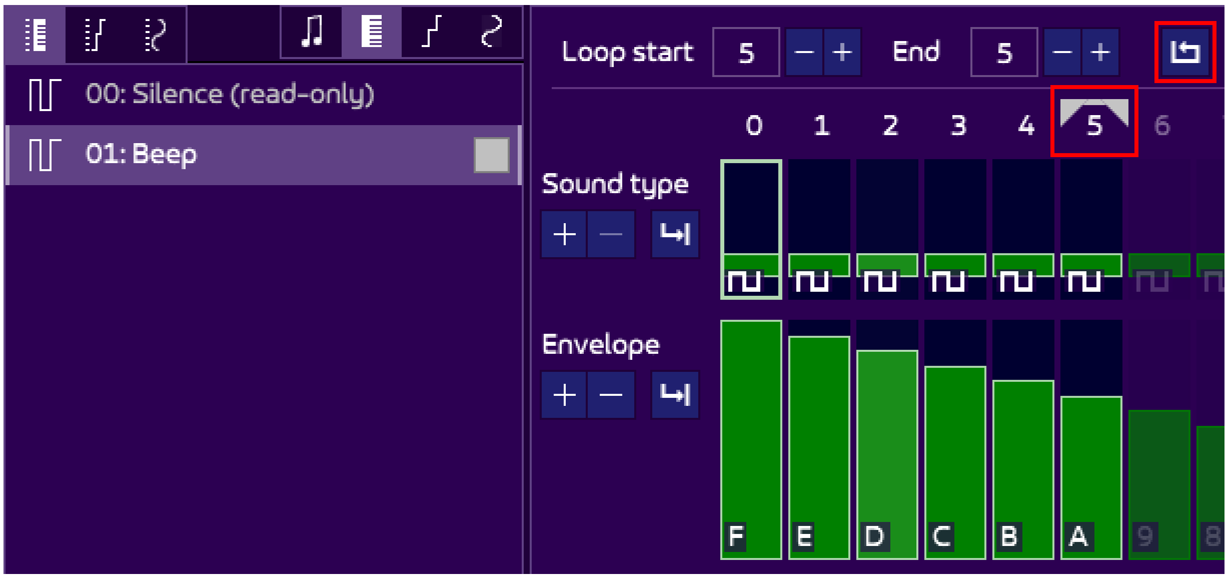
Using arpeggios

In this tutorial, we'll see how to create arpeggios and how to use their related effects in the patterns.

> Before reading this, you should have a basic understanding of AT3, which this tutorial will help you get! You should also read about the arpeggio editor.

Arpeggios are among the most expressive tricks in sound-chip music. Not only can it simulate several notes in one channel, it almost gives a composer its personality. Tim Follins, David Whittaker, Dave Rogers all have their trademark arpeggios! Now it's your turn.

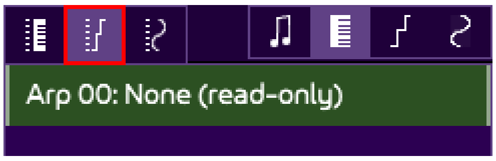
Let's dive directly into the matter. Start AT3. It opens with a new empty song. A first "Beep" sound is present. Open it and make it loop on the sixth bar:



This simple sound will be the starting point upon which we will make our tests.

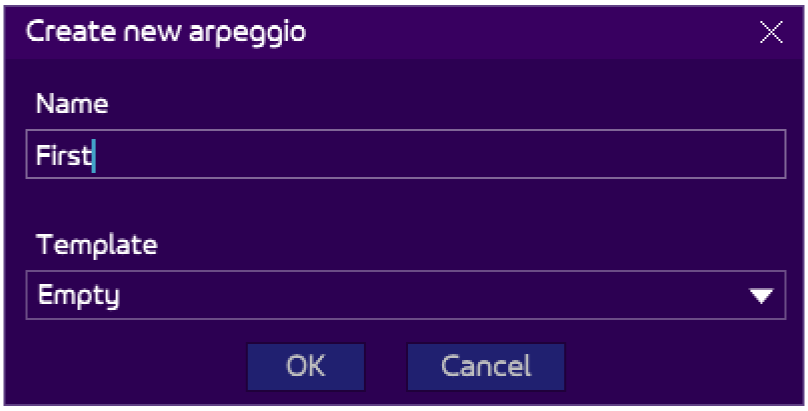
Creating a first arpeggio

On the left of the screen is the list of instruments (as you can see above), arpeggios and pitches. Press the second icon from the left to open the arpeggio list (AL). You can also press shift + F4:


Arp 00: None (read-only)

There is only one arpeggio, called “None”. It is always present and is used when you want to stop arpeggios, but we’ll see that later.

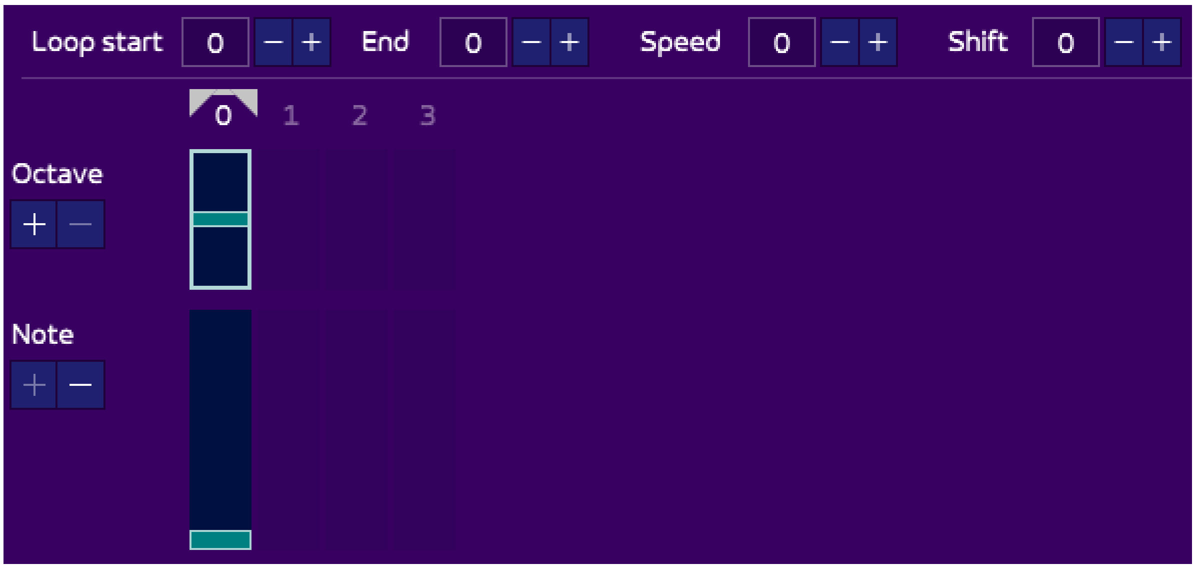
Just like creating a new instrument, press + at the bottom of the list. A dialog opens, prompting you to name your new arpeggio and select a template. Do as the following, and press OK:


Create new arpeggio

Name: First

Template: Empty

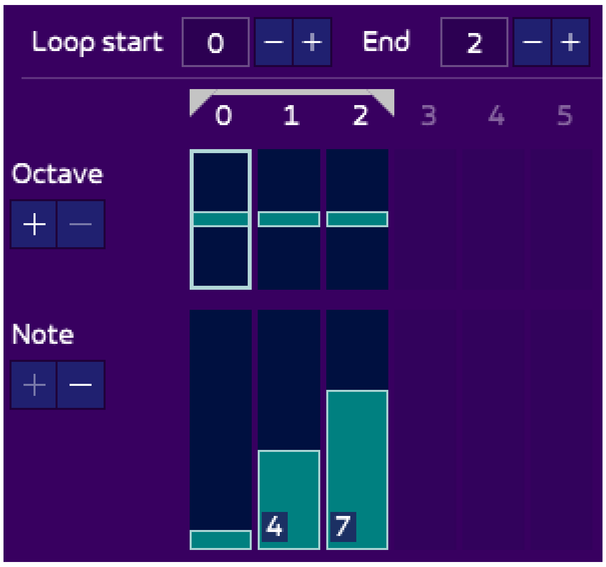
“First” now appears in the arpeggio list. Let’s edit it: double-click on it, and the Arpeggio Editor will appear in the center of the screen (you can also press F4 to open the currently selected arpeggio. Note that opening the editor will automatically opens the arpeggio list too, which is a handy shortcut!):


Loop start 0 End 0 Speed 0 Shift 0

Since the template we selected was “empty”, the arpeggio contains nothing. Only one column, and no value in it (0s are not written to have a clearer interface). *Octave* adds full octave to the played notes, and *Note* adds semi-tones to it.

You may not know musical theory (I certainly don't!), but by adding 4, then 7 semi-tones, you get a major chord. Let's do that! You should know how to add values in the editors (check this if that's not the case). Let's do it the easy way by using the mouse.

Place your mouse above the *Note* row, go to the *second and third rows* (named 1 and 2) and, by using ctrl + left-click, modify/add the values 4 and 7. The row 0 should remain as-is. You should have something like this:



Note two things:

- The loop end automatically set itself to 2, because we actually added values that weren't there, so AT figured out you probably wanted to include them in your arpeggio.
- The loop is always on and cannot be turned off. Contrary to instruments, arpeggios (and pitches) always loop!

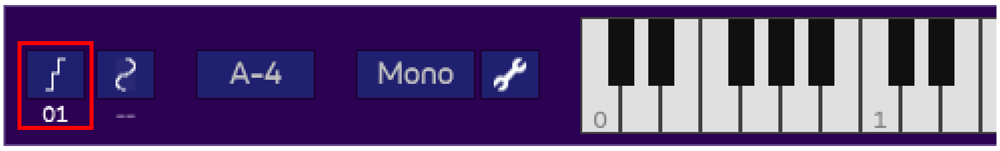
Testing the arpeggio

Let's hear our creation! We need a sound upon which the arpeggio is played. The "Beep" instrument should be selected automatically when launching AT, so by playing with your MIDI keyboard, or your computer keyboard (QWERTY or ZXCVBN), you should directly hear it! Press escape to stop it.

0:00 / 0:03

How great! "First" is not very evocative, so right-click on it in the AL, select "Rename", and type "Major" (because, no surprise, it is a major chord).

Note: your arpeggio is used when testing the sound because it is selected in the arpeggio list, which you can also see in the Test Area below, with a "01" just below the arpeggio icon):

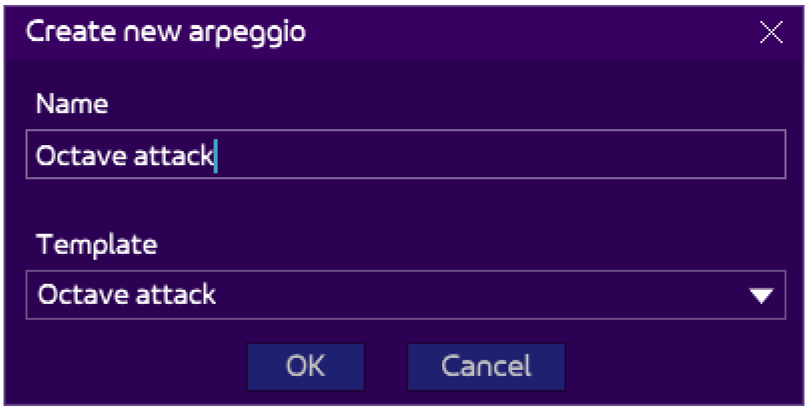


More on the test area here.

A second arpeggio

The arpeggio we just did is perfect for accompaniment. But we could make another one for a melody. The most obvious one would be an "octave attack". That's nice, AT3 provides a template for this!

Either right-click on the last arpeggio on the AL and select "Create at the bottom", or click on the + icon at the bottom of the list. A dialog opens for our new arpeggio. Name it "Octave attack" and select the like-named template:

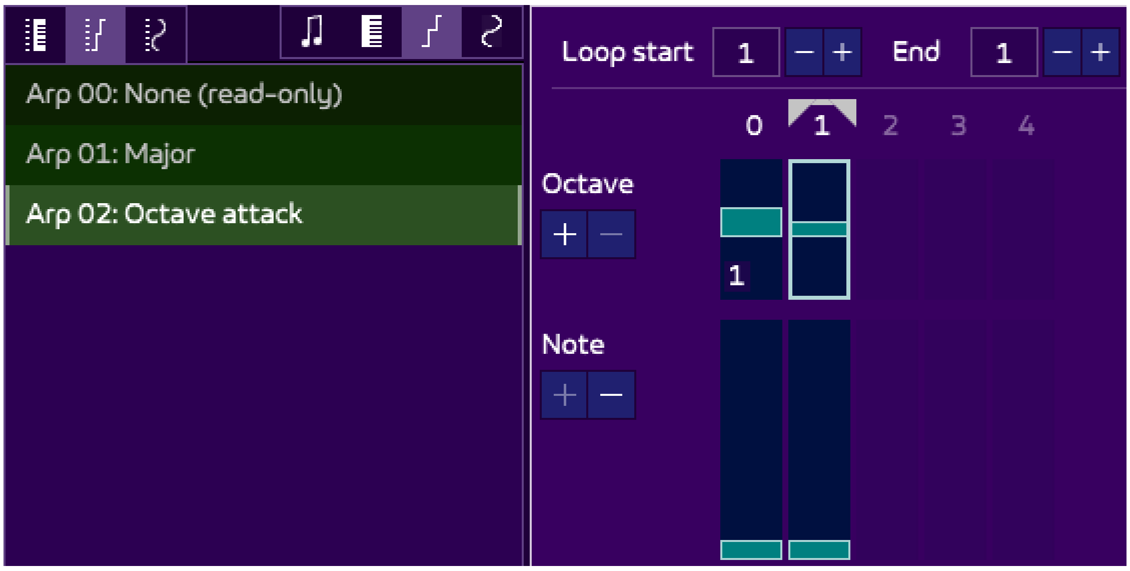


Create new arpeggio

Name: Octave attack

Template: Octave attack

Opens the newly created arpeggio:



Arp 00: None (read-only)
Arp 01: Major
Arp 02: Octave attack

Loop start 1 End 1

Playing with it sounds like this:

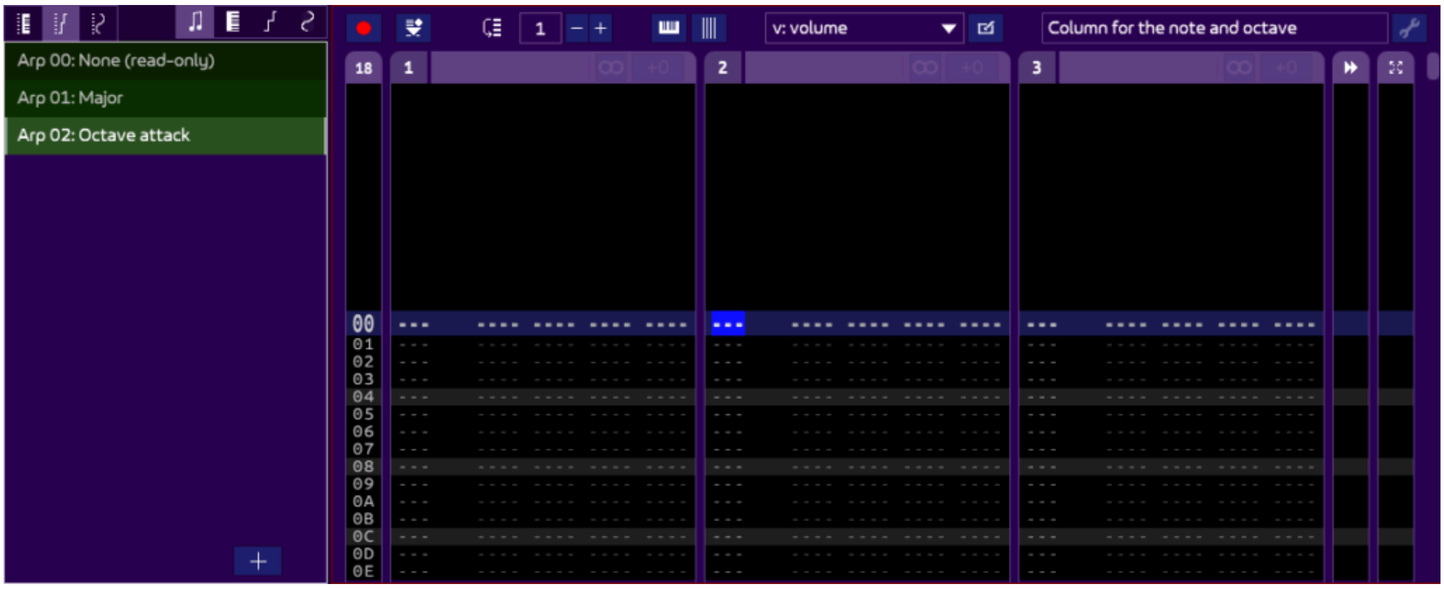
0:00 / 0:02

As you can see and hear, the first frame has an octave higher than the rest, and the arpeggio then loops on the second row, which has no particular data. The sound thus remains bare. We've just added an arpeggio that only modifies the start of the sound and lets the remaining unmodified.

Using the arpeggios in a song

All this is nice, but we want to create a song with these arpeggios! Like pitches, arpeggios are used via the effects columns of the Pattern Viewer (explained thoroughly here). By now you must know how to write notes and some effects (as seen in the previous tutorial), but arpeggio effects still remain a mystery. Let's clear this!

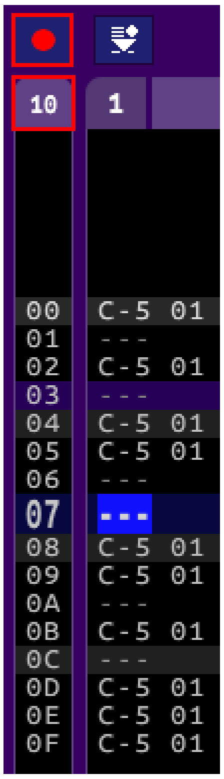
Go to the PV by pressing F2. We are in front on an empty pattern.



Notice in the AL on the left that each arpeggio has a number in front of it:

- `00` for None
- `01` for Major
- `02` for Octave attack

These numbers indicate how arpeggios are referred to in the musical score. Let's write a simple arrangement with these notes, on the first channel: (Make sure that Record is on: ctrl + space / ctrl + r on Mac, or click on the Record button).



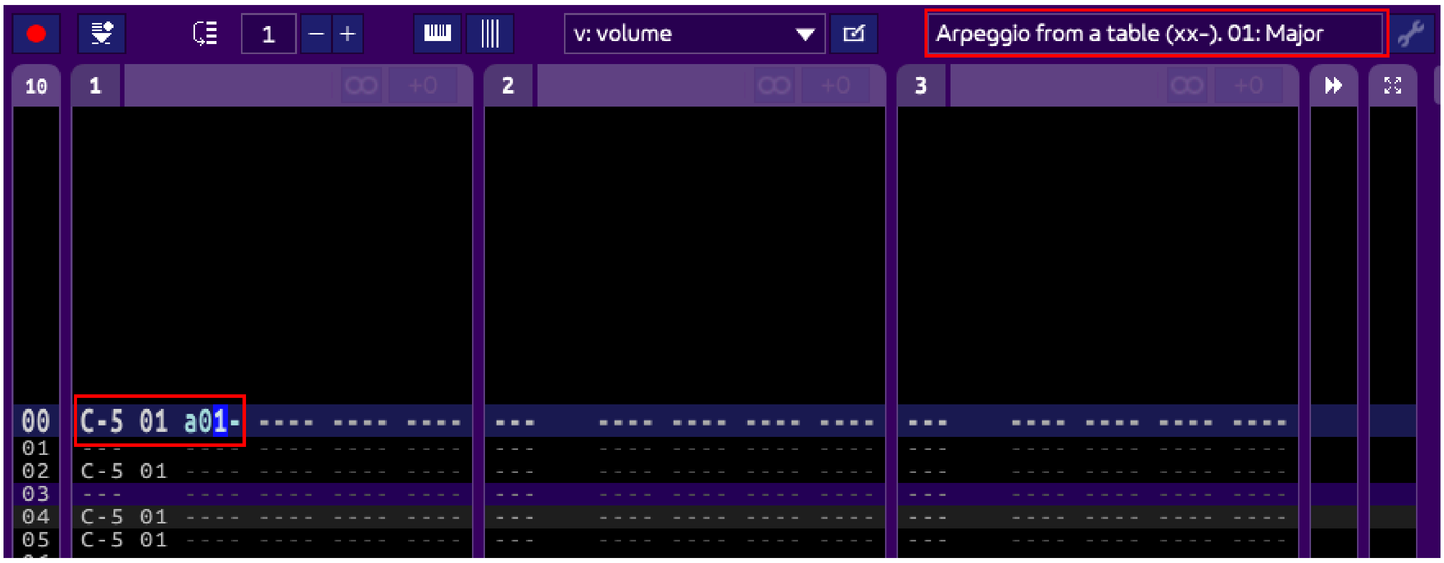
```
10    1
00    C-5 01
01    ---
02    C-5 01
03    ---
04    C-5 01
05    C-5 01
06    ---
07    ---
08    C-5 01
09    C-5 01
0A    ---
0B    C-5 01
0C    ---
0D    C-5 01
0E    C-5 01
0F    C-5 01
```

Also, reduce the position size by clicking on its height number at the top-left. Let's set it to 10 (hex).

Play the song, it should sound like this:

0:00 / 0:01

Ok, not very exciting. Now move up to the first line and add the following effect:



v: volume

Arpeggio from a table (xx-). 01: Major

```
00    C-5 01 a01-
01    ---
02    C-5 01
03    ---
04    C-5 01
05    C-5 01
```

Tip: if you go back to the first line where my cursor is, you can see at the top-right an information about what is under the cursor: an "arpeggio from table effect", with the currently set: 01: Major.

Play again! This time it sounds pretty cool (and reminds me a bit of a famous CPC song from a famous game made by a famous composer (hi BSC!)):

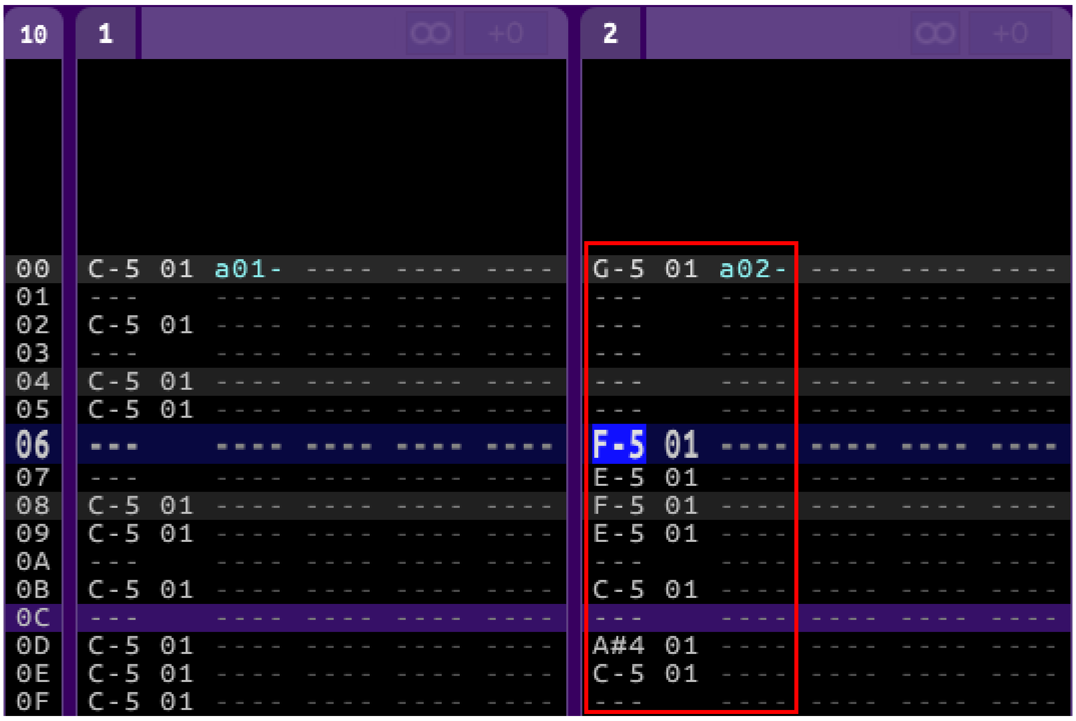
0:00 / 0:01

Notice how even though we only added one arpeggio, all the notes are using it. That's because expressions (arpeggios and pitches) **remain** on the track they are used, and **restarts** every time a note is played. So you don't have to copy/paste the effect on every note, which is very convenient.

This also means that you may want to stop them if you don't want to use them anymore! How? Simple: simply use the arpeggio `00` ("None").

Adding a melody

It's now time to use the second arpeggio we created. Let's write a small melody on the second track using the "Octave attack" arpeggio:



```
10   1                               +0   2                               +0

00   C-5 01 a01- ---- ---- ----           G-5 01 a02- ---- ---- ----
01   ---        ---- ---- ---- ----       ---        ---- ---- ---- ----
02   C-5 01 ---- ---- ---- ----           ---        ---- ---- ---- ----
03   ---        ---- ---- ---- ----       ---        ---- ---- ---- ----
04   C-5 01 ---- ---- ---- ----           ---        ---- ---- ---- ----
05   C-5 01 ---- ---- ---- ----           ---        ---- ---- ---- ----
06   ---        ---- ---- ---- ----       F-5 01 ---- ---- ---- ----
07   ---        ---- ---- ---- ----       E-5 01 ---- ---- ---- ----
08   C-5 01 ---- ---- ---- ----           F-5 01 ---- ---- ---- ----
09   C-5 01 ---- ---- ---- ----           E-5 01 ---- ---- ---- ----
0A   ---        ---- ---- ---- ----       ---        ---- ---- ---- ----
0B   C-5 01 ---- ---- ---- ----           C-5 01 ---- ---- ---- ----
0C   ---        ---- ---- ---- ----       ---        ---- ---- ---- ----
0D   C-5 01 ---- ---- ---- ----           A#4 01 ---- ---- ---- ----
0E   C-5 01 ---- ---- ---- ----           C-5 01 ---- ---- ---- ----
0F   C-5 01 ---- ---- ---- ----           ---        ---- ---- ---- ----
```

0:00 / 0:01

… which also reminds me of yet another famous melody rendered on CPC by another famous musician (hi Kangaroo!).

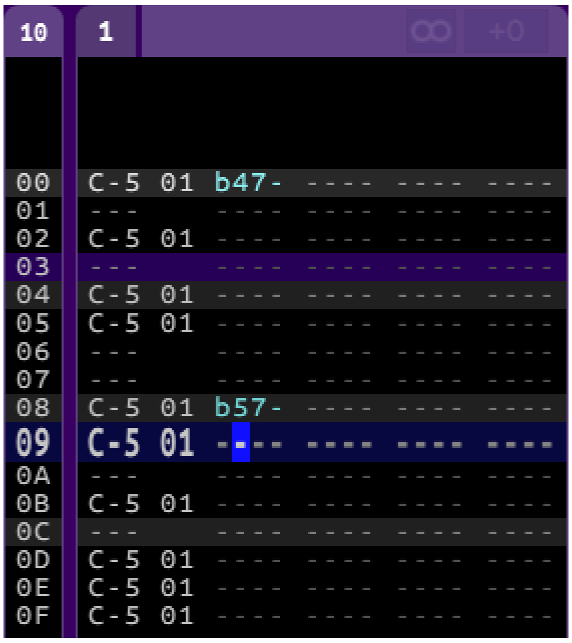
Inline arpeggios

Creating arpeggios via the arpeggio editor is nice, but you can't help but wondering if there wouldn't be a way to create simple arpeggio in a simpler way. Well… yes! It's called "inline arpeggios". It means you can declare an arpeggio directly in the pattern.

There are two awesome effects for this:

- 3-note arpeggio (effect b)
- 4-note arpeggio (effect c)

The principle is that all the semi-tones to add are written in the effect parameter. Our "Major" chord arpeggio is composed of 0-4-7 semi-tones. Let's modify our first track by replacing the `a01` effect with these:



```
10   1                    +0

00   C-5 01 b47- ---- ---- ----
01   ---    ---- ---- ---- ----
02   C-5 01 ---- ---- ---- ----
03   ---    ---- ---- ---- ----
04   C-5 01 ---- ---- ---- ----
05   C-5 01 ---- ---- ---- ----
06   ---    ---- ---- ---- ----
07   ---    ---- ---- ---- ----
08   C-5 01 b57- ---- ---- ----
09   C-5 01 ---- ---- ---- ----
0A   ---    ---- ---- ---- ----
0B   C-5 01 ---- ---- ---- ----
0C   ---    ---- ---- ---- ----
0D   C-5 01 ---- ---- ---- ----
0E   C-5 01 ---- ---- ---- ----
0F   C-5 01 ---- ---- ---- ----
```

Without the melody, it sounds like this:

0:00 / 0:01

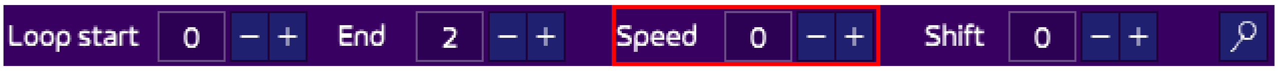
Whoo, it's great! But then you ask: since we did in one line what we bothered to do with bars in the AE, why bother with the AE at all!?? A few reasons, my friend!

- Inline arpeggios are limited to three notes ("b" effect) or four notes ("c" effect). The base note (0), implicit, is always the first, followed by the ones written in the pattern (47 or 57 in the example). You can't change this order!

- The semi-tones range is limited from 0 to F, which is more than one octave, but is still not much. Also, there are no negative semi-tones.
- The speed is always the maximum. Not a huge deal, see below for a neat trick.
- The loop is forced on all the 3 or 4 notes the inline arpeggio contains. So even a simple arpeggio like “Octave attack” cannot be reproduced.
- On a low-level technical note, most exported format (AKG, AKM) will convert inline arpeggios to arpeggio tables, which are limited to 255 at best. If you create many *different* inline arpeggios, the export will fail (don’t worry, I never saw anyone reaching this limitation!).

Changing the speed

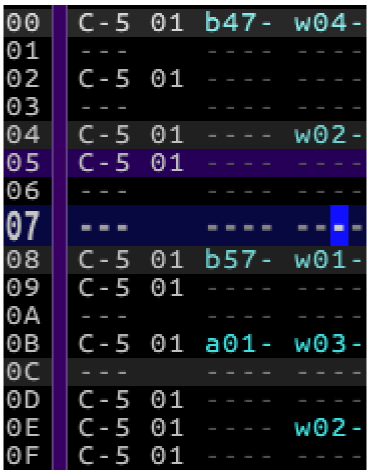
You know that arpeggios, just like instruments and pitches, have a speed parameter in the editor, going from 0 (fastest) to 255 (slowest):



```
Loop start 0 - +   End 2 - +   Speed 0 - +   Shift 0 - +
```

This is great, but what if you want, once or at a few specific moments, to change its speed? One possibility is to duplicate the arpeggio only to set a new speed. It is actually a good solution if you use this new arpeggio a lot, but it is inefficient in most cases.

Well, AT got you covered! There is an effect called Set arpeggio speed (“w” effect) you can use to set the speed of the arpeggio that is just *before in the same line*. Here is an example, using both inline and editor arpeggios:



```
00  C-5 01 b47- w04-
01  ---    ---- ----
02  C-5 01 ---- ----
03  ---    ---- ----
04  C-5 01 ---- w02-
05  C-5 01 ---- ----
06  ---    ---- ----
07  ---    ---- ----
08  C-5 01 b57- w01-
09  C-5 01 ---- ----
0A  ---    ---- ----
0B  C-5 01 a01- w03-
0C  ---    ---- ----
0D  C-5 01 ---- ----
0E  C-5 01 ---- w02-
0F  C-5 01 ---- ----
```

At slower speed, it sounds like this:

0:00 / 0:05

A lot of things can be said on these few lines of score:

- The “w” effect only works on the line it is put on. The next time the arpeggio is used, it will use its default speed. For example, line `00` has an arpeggio speed of `04`, but line `02` comes back to speed `00`, as it is the default speed of inline arpeggios.
- You can use the “w” effect even if no arpeggio is declared (see line `04` and `0E`). The triggered arpeggio will use this new speed.
- The “w” effect works on all kind of arpeggios! Inline and normal ones (see line `0B` and `0E`).

Two more technical things to bear in mind:

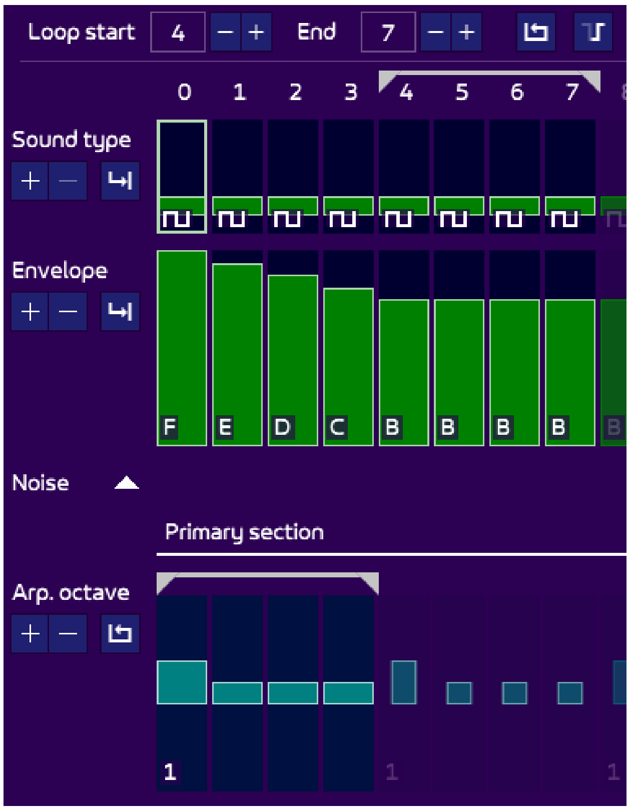
- If you start using this effect too much on one specific inline arpeggio, it is probably more handy to replace the inline arpeggios with a “real” arpeggio using the arpeggio editor.
- The same for a “real” arpeggio: you could probably duplicate it and change its speed.

Double arpeggios

This is an slightly advanced topic, but soon you won’t stop using it when you hear what can be done with it. You know there are two ways of using arpeggios:

- Via effects as explained above (arpeggio table, inline arpeggio).
- Inside the instruments themselves.

Now, what would happen if you combine both? Would AT explode? Certainly not. The values will **add up**! The following example will use an instrument that loops its own arpeggio. It looks and sounds like this:



0:00 / 0:05

The octave loops till the end of the sound, which itself loops (notice the use of the auto-spread feature in the *Arp.octave* row).

Now, if we create an arpeggio made of a simple major chord, with a speed of 4, and plays it via an arpeggio table effect, plus the sound above, we would get this:

0:00 / 0:07

Wow, isn't that great? The looping octave of the instrument is added to the simple basic chord of the arpeggio. Yet the sound is much richer. How come? It's simple: it's because both the instrument octave and the arpeggio have a different cycle (made possible by the size of their loop, and their different speed), so the sound appears much longer than what it actually is.

Experience shows that it is more flexible to have the instrument arpeggio very simple (like an octave change), and have the more complex arpeggio being done via arpeggio table.

You can experiment with such technique by playing your pattern, and tweaking the instrument speed/loop and the arpeggio table speed/size. There are unlimited possibilities!

Generating arpeggios

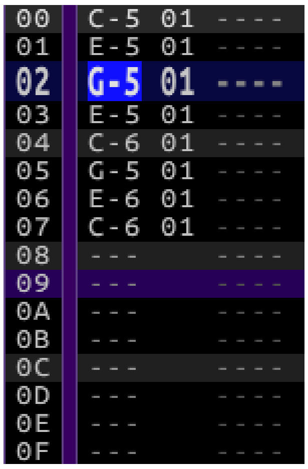
This is a slightly more advanced topic, but I feel you're ready for it :). Sometimes the musician in you has the notes of the arpeggio in mind, but "writing" them in the arpeggio editor via bars or semi-tones is tedious to you. Why decompose them in semi-notes when you can "hear" them?

Well, there is a great feature in AT3 that will help you create arpeggios from notes, and it will also enable you to create very complex arpeggio in a blink of an eye.

The principle is simple: we will write the notes in the PV for them to be **generated** into an arpeggio.

Delete the first track with the accompaniment, because we will write something better.

Now type the following notes:

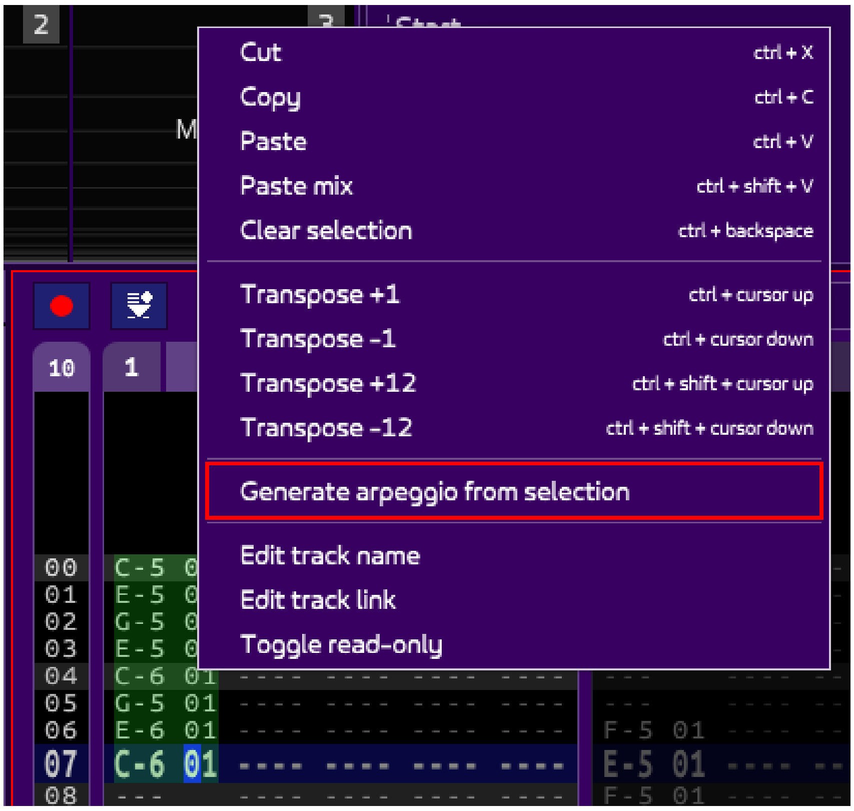


```
00  C-5 01 ----
01  E-5 01 ----
02  G-5 01 ----
03  E-5 01 ----
04  C-6 01 ----
05  G-5 01 ----
06  E-6 01 ----
07  C-6 01 ----
08  ---    ----
09  ---    ----
0A  ---    ----
0B  ---    ----
0C  ---    ----
0D  ---    ----
0E  ---    ----
0F  ---    ----
```

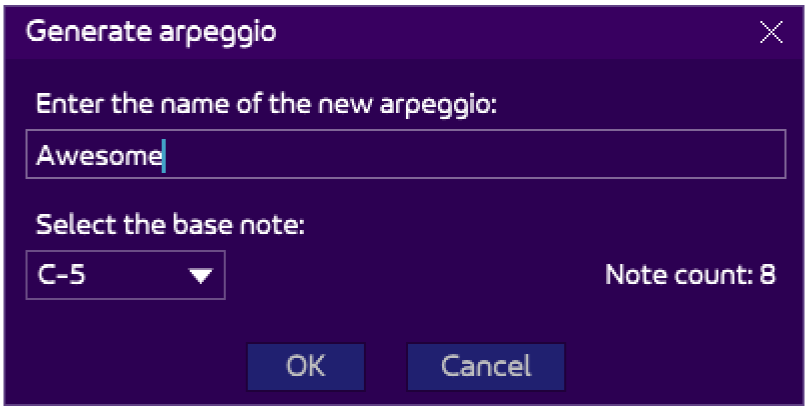
They should sound like this:

0:00 / 0:01

Nothing fancy, but wait! Select all of them, right-click on the selection and select "Generate arpeggio from selection"


Cut ctrl + X
Copy ctrl + C
Paste ctrl + V
Paste mix ctrl + shift + V
Clear selection ctrl + backspace
Transpose +1 ctrl + cursor up
Transpose -1 ctrl + cursor down
Transpose +12 ctrl + shift + cursor up
Transpose -12 ctrl + shift + cursor down
Generate arpeggio from selection
Edit track name
Edit track link
Toggle read-only

A pop-up shows:


Generate arpeggio

Enter the name of the new arpeggio:
Awesome

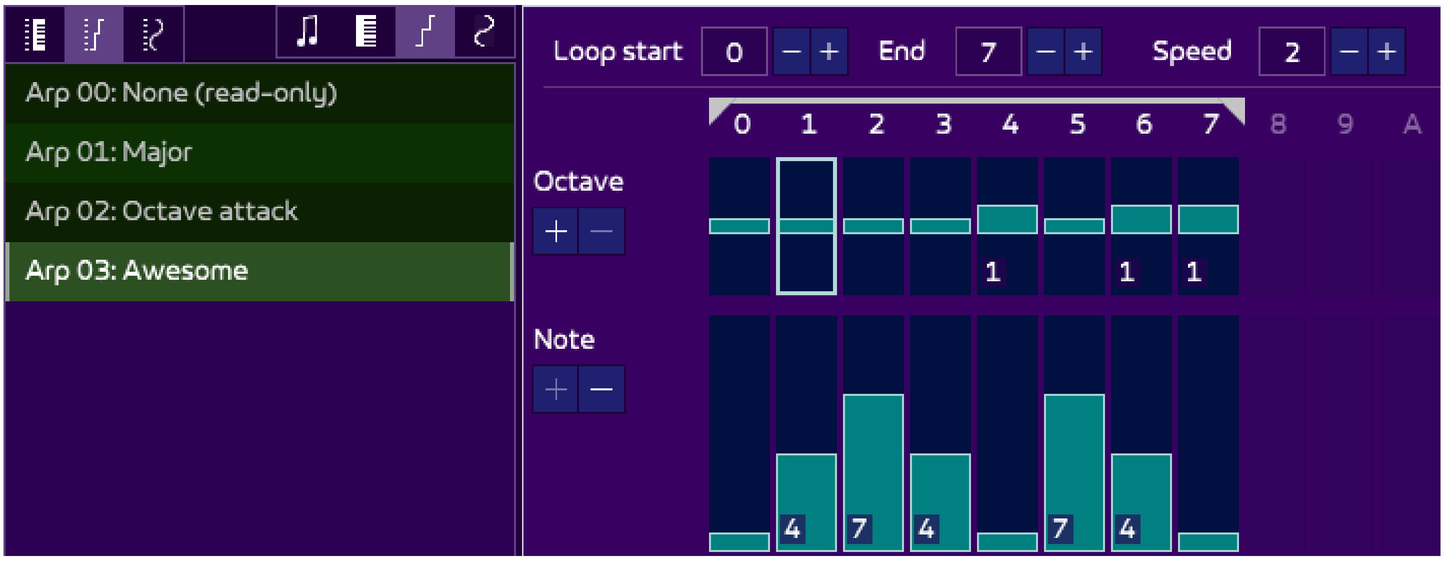
Select the base note:
C-5

Note count: 8

OK Cancel

Enter the name “Awesome” (because it is), and let C-5 as the base note (it will be a reference note of our new arpeggio. Most of the time, leave this parameter alone). Click OK!

A new arpeggio has been generated, and it is “Awesome”!



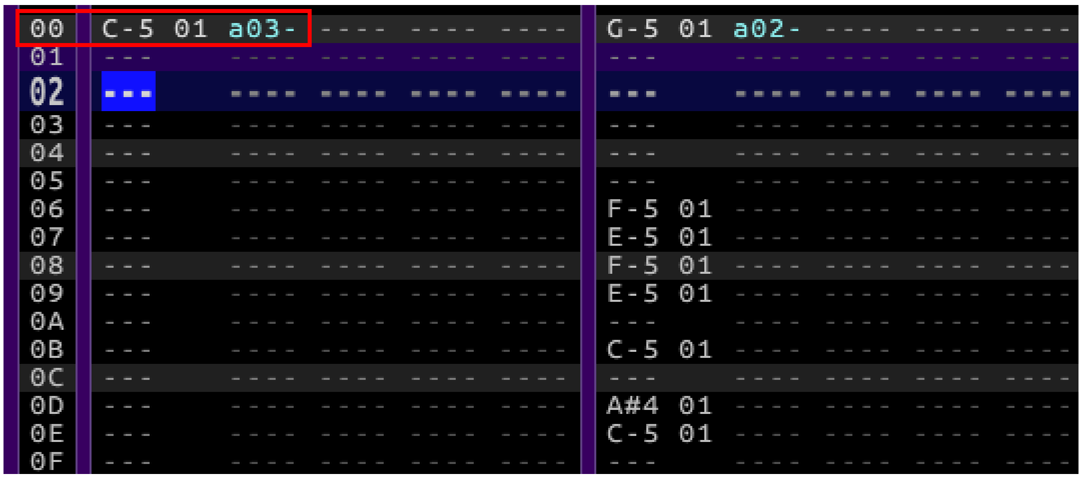
Arp 00: None (read-only)
Arp 01: Major
Arp 02: Octave attack
Arp 03: Awesome

Loop start 0 End 7 Speed 2

Reduce its speed to 5 to hear it better and have fun with it. Don't you recognize the notes we've just entered?

0:00 / 0:07

Now let's use it in our song! Set its speed to 2, erase the first track notes again, and simply use the Awesome arpeggio:



```
00  C-5 01 a03- ---- ---- ----   G-5 01 a02- ---- ---- ----
01  ---        ---- ---- ---- ----   ---        ---- ---- ---- ----
02  ---        ---- ---- ---- ----   ---        ---- ---- ---- ----
03  ---        ---- ---- ---- ----   ---        ---- ---- ---- ----
04  ---        ---- ---- ---- ----   ---        ---- ---- ---- ----
05  ---        ---- ---- ---- ----   ---        ---- ---- ---- ----
06  ---        ---- ---- ---- ----   F-5 01 ---- ---- ---- ----
07  ---        ---- ---- ---- ----   E-5 01 ---- ---- ---- ----
08  ---        ---- ---- ---- ----   F-5 01 ---- ---- ---- ----
09  ---        ---- ---- ---- ----   E-5 01 ---- ---- ---- ----
0A  ---        ---- ---- ---- ----   ---        ---- ---- ---- ----
0B  ---        ---- ---- ---- ----   C-5 01 ---- ---- ---- ----
0C  ---        ---- ---- ---- ----   ---        ---- ---- ---- ----
0D  ---        ---- ---- ---- ----   A#4 01 ---- ---- ---- ----
0E  ---        ---- ---- ---- ----   C-5 01 ---- ---- ---- ----
0F  ---        ---- ---- ---- ----   ---        ---- ---- ---- ----
```

Let's hear it!

0:00 / 0:03

Isn't it awesome? Now you can write simple to hyper-rich arpeggios without having to draw these damned bars in the editor! Doesn't this arpeggio has a "Tim Follin" vibe, mmmh? It sure does, which, I think, is a pretty good sign as to what AT can do for you. Now go create your own arpeggio signature!

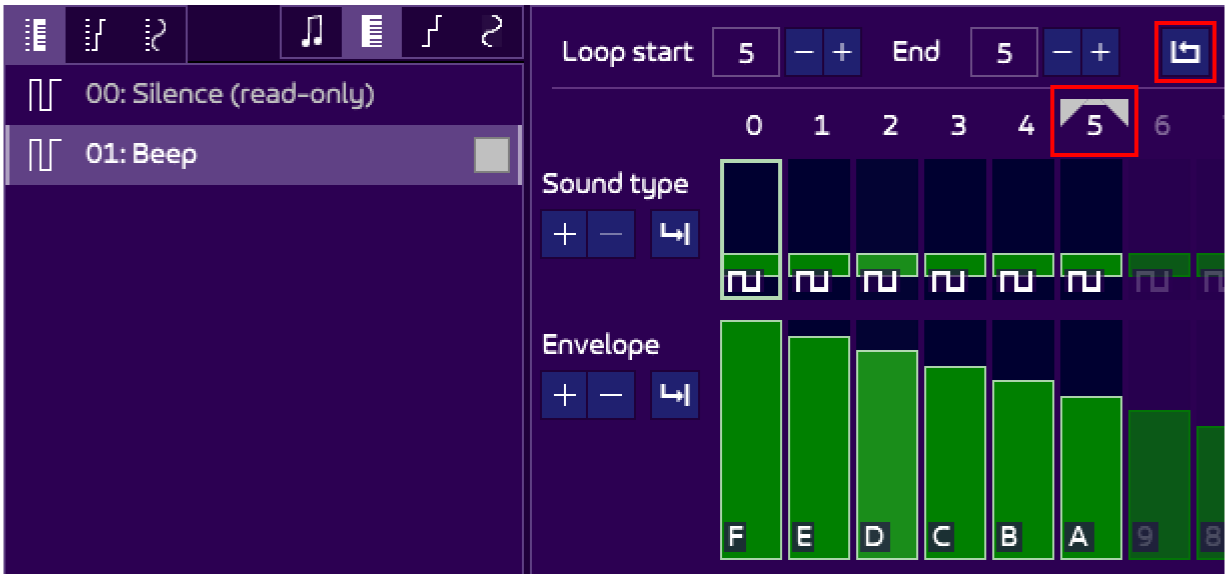
Using pitches

Just like you learned with arpeggios, this tutorial will show you how to create pitches and use the related effects in the patterns.

> Before reading this, you should have a basic understanding of AT3, which this tutorial will help you get! You should also read about the pitch editor.

Besides volume effects, pitches are probably the most used effects in the sound-chip world.

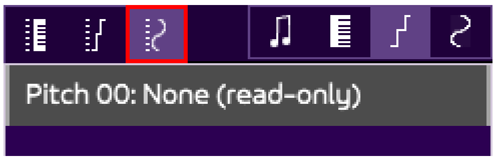
Let’s create some. Start AT3. It opens with a new empty song. A first “Beep” sound is present. Just like with the arpeggio tutorial, open it and make it loop on the sixth bar:



This simple sound will be the starting point upon which we will make our tests.

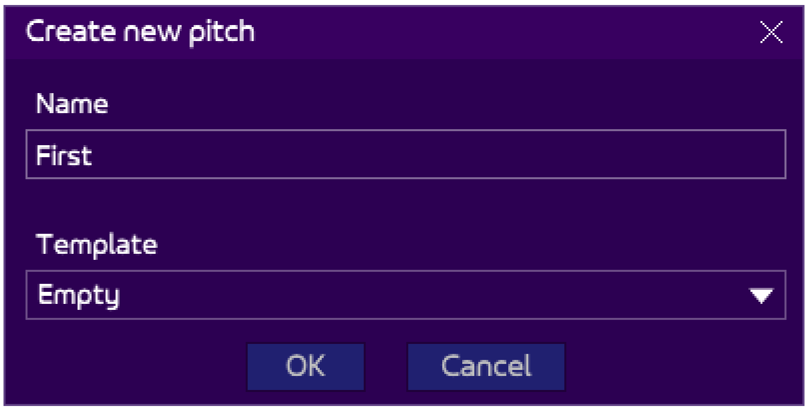
Creating a first pitch

On the left of the screen is the list of instruments (as you can see above), arpeggios and pitches. Press the third icon from the left to open the pitch list (PL). You can also press shift + F5:


Pitch 00: None (read-only)

Analogue to the arpeggio, there is only one pitch by default, called “None”. It is always present and is used when you want to stop pitches, but we’ll see that later.

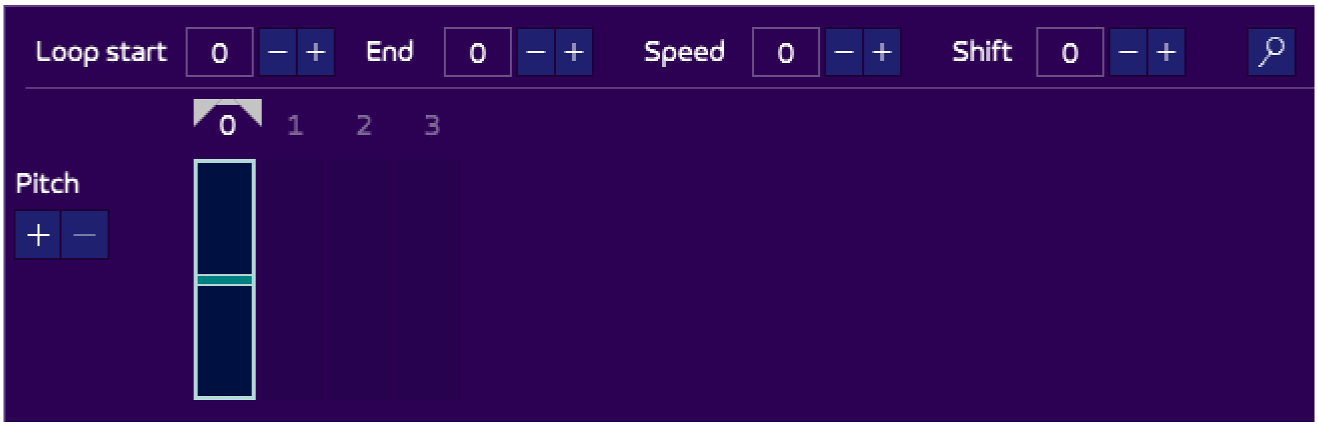
Click on the + at the bottom of the pitch list, which opens a prompt. Enter the fields as the following and press OK to create the pitch:


Create new pitch

Name: First

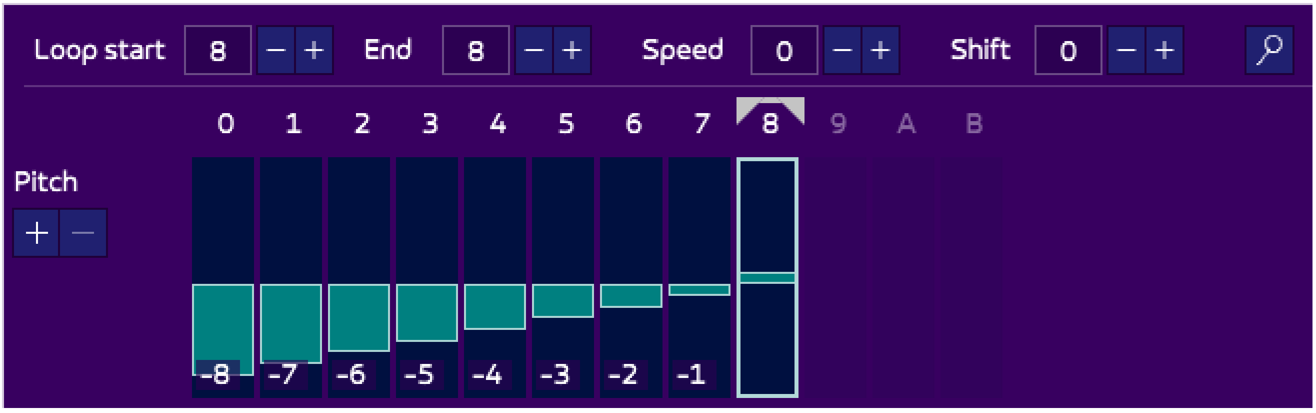
Template: Empty

“First” appears in the list. Double-click on it, and the Pitch Editor will appear in the center of the screen (you can also press F5 to open the currently selected pitch. Opening the editor will also automatically open the list, which is handy!):


Loop start 0 End 0 Speed 0 Shift 0

There is nothing really fancy here. Only one column, and no value in it (0s are not written to have a clearer interface). The value of a pitch is added to the frequency of the sound (to be more precise, it is added to its *period*). We can thus simulate vibratos, slides, or go crazy with strange effects.

By now you should be used to using the bar editors of AT3. Use any mean you know (or one described here) to modify the pitch such as to get a nice slide sound during a few bars, with a loop at the last row (which should have a value of 0, else the sound will be off-key):



By using the keyboard, you can directly play the sound (and escape to stop it):

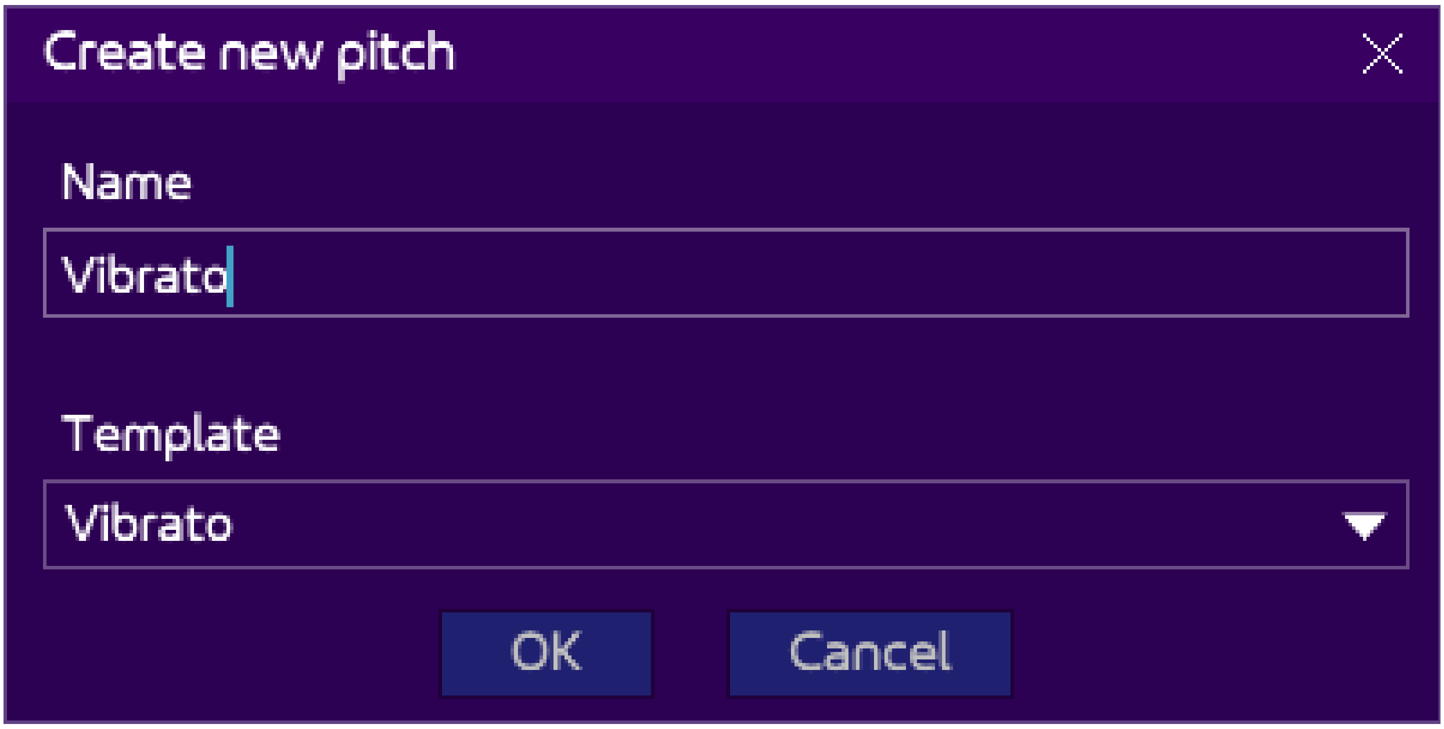
0:00 / 0:02

You can rename the pitch to “Low attack” for example.

A second pitch

We will create a second pitch from the AT3 templates (which, actually, “Low attack” belongs to!).

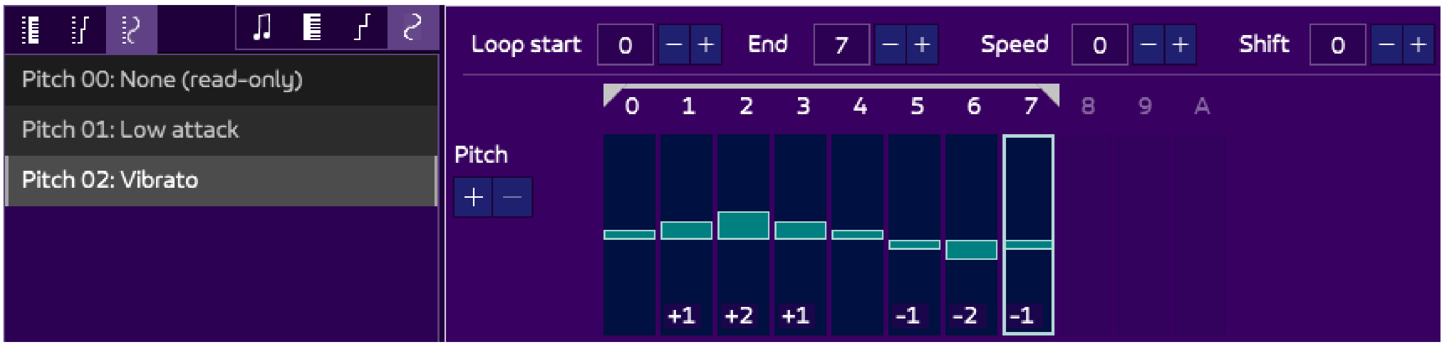
You know the drill. Right-click an item on the PL and select “Creates at the bottom”, or click on the + icon at the bottom of the list. A dialog opens for our new pitch. Name it “Vibrato” and select the like-named template:


Create new pitch

Name: Vibrato

Template: Vibrato

Open the newly created pitch:


Pitch 00: None (read-only)
Pitch 01: Low attack
Pitch 02: Vibrato

Loop start 0 End 7 Speed 0 Shift 0

It sounds like this:

0:00 / 0:02

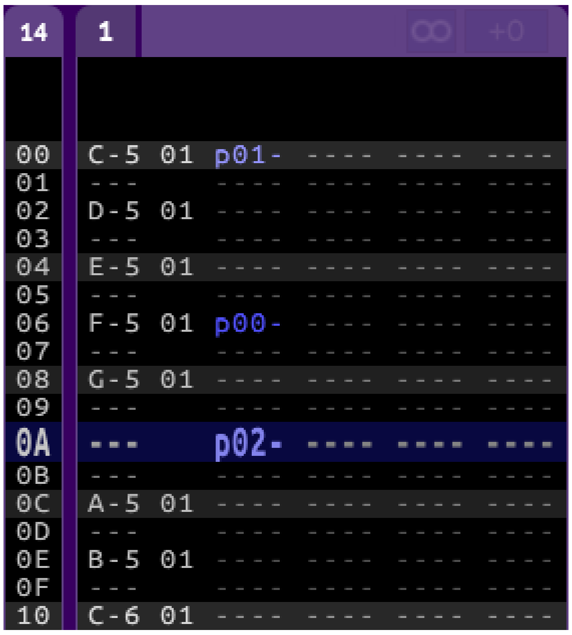
Contrary to the first pitch which loops on one one bar (the last one), this vibrato loops on all its length. But it is only an example, don't hesitate to change the loop and test if it sounds better to you ears!

Using the pitches in a song

The principle is the same as with arpeggios, but the effects are different. There are no "inline pitches", only "pitch table". But there are more pitch effects, which we're going to see just after… Finally, as a bonus, we're going to mix arpeggio and pitch for an even richer sound.

The pitch table effect is the "p" effect. It is followed by the pitch number, which you can see listed in the PL, just besides the pitch you are interested in (`00` to stop the pitch, `01` for Low attack, `02` for Vibrato).

Writing a melody using the pitch is straightforward and follow the same rules as the arpeggios:



```
14  1                               ∞   +0

00  C-5 01 p01- ---- ---- ----
01  ---        ---- ---- ---- ----
02  D-5 01 ---- ---- ---- ----
03  ---        ---- ---- ---- ----
04  E-5 01 ---- ---- ---- ----
05  ---        ---- ---- ---- ----
06  F-5 01 p00- ---- ---- ----
07  ---        ---- ---- ---- ----
08  G-5 01 ---- ---- ---- ----
09  ---        ---- ---- ---- ----
0A  ---        p02- ---- ---- ----
0B  ---        ---- ---- ---- ----
0C  A-5 01 ---- ---- ---- ----
0D  ---        ---- ---- ---- ----
0E  B-5 01 ---- ---- ---- ----
0F  ---        ---- ---- ---- ----
10  C-6 01 ---- ---- ---- ----
```

It will sound like this:

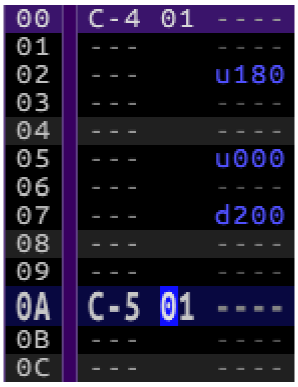
0:00 / 0:06

Interesting facts:

- Pitch `01` is used on the line `00`, and the following notes on lines `02` and `04` are using it even though no pitch effect is present. The pitch table is restarted on each new note, and there is no need to write the effect every time. It continues on the track as long as you didn't tell it to shut up.
- Pitch `00` is used on line `06`, and this effectively stops any pitch. The notes on line `06` and `08` have thus no pitch table.
- Pitch `02` is used on line `0A`. Interestingly, it is not declared besides a note, but on an empty space. Yet, it is added to the currently played instrument. So you can actually start a pitch anywhere.

Pitch up/down

This effect has nothing to do with the pitch table, but it is a pitch effect nonetheless! It simply increases/decreases the pitch over time.


```
00  C-4 01 ----
01  ---    ----
02  ---    u180
03  ---    ----
04  ---    ----
05  ---    u000
06  ---    ----
07  ---    d200
08  ---    ----
09  ---    ----
0A  C-5 01 ----
0B  ---    ----
0C  ---    ----
```

It sounds like this:

0:00 / 0:04

Both pitch up ("u") and pitch down ("d") effects are followed by a speed, with a value of `000` to stop the pitch. Notice how the last note in line `0A` stops the pitch too, simply by its presence. Pitch up/down effects are cancelled every time a new note appears.

Pitch table and up/down!

The title says it all. The pitch up/down effects can be mixed with the pitch table. Take the example just above but add the Vibrato pitch table (`p02`) on the first note (with some reduced pitch up/down value to hear the vibrato better):

0:00 / 0:04

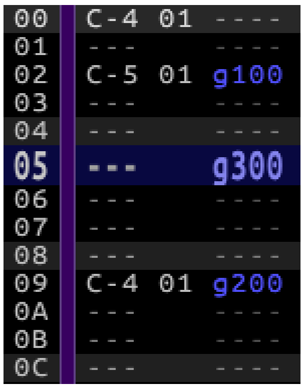
Now that's two layers of pitch, which I think is pretty powerful!

If you think the pitch up/down are not fast enough, you can use the fast pitch up/down effect.

Pitch glide

You may have had fun with the pitch up/down effects already. One drawback is that it is hard to reach a specific note. You can tweak the speed but it's boring, and you often end up being off-key. Pitch glide ("g" effect) to the rescue! It is a nice effect to link two notes via a pitch. It's like a pitch, but with a destination.

It must be first called besides a note to set the destination:



```
00  C-4 01 ----
01  ---    ----
02  C-5 01 g100
03  ---    ----
04  ---    ----
05  ---    g300
06  ---    ----
07  ---    ----
08  ---    ----
09  C-4 01 g200
0A  ---    ----
0B  ---    ----
0C  ---    ----
```

It sounds like this:

0:00 / 0:05

- The first note is C-4, plays normally.
- The second note is C-5, but with a glide effect. The C-5 note is not played, C-4 is continued, but pitches up till it reaches C-5.
- At line 05, a glide effect is still present, but alone this time. It is possible to use it this way when a glide is already started: you can modify the speed of the glide on the fly.
- Somewhere between lines 05 and 09, the C-5 note is reached by the glide: it stops gliding!
- At line 09, another new note and a glide effect: this time, the pitch goes down.

One advantage is that you don't have to define a direction, the glide goes up or down according to the destination note. Three drawbacks:

- This effect is slightly more costly in CPU than a pitch up/down (AKG).
- It is not available in AKM format.
- Just like all pitches, be aware that such effect has a cost in size for streamed formats (AKY).

Mixing arpeggios and pitches

As a bonus, another example. AT3 makes it possible to mix arpeggios and pitches. You can for example have an arpeggiated melody, plus a vibrato, with a pitch down at the end for good measure (pun intended):

0:00 / 0:05

David Whittaker (hear the wonderful “Beyond the Ice Palace” songs) and Allister Brimble (the Dizzy series) both use a lot of such effects. The possibilities are quite enormous! Have fun!

Creating a full song (video)

What’s a better way to learn that watching me create a song before your very eyes? Well, here is it for you!

I chose Lop Ears by Andy Severn, because it is both beautiful and quite simple. I hope you will enjoy that kind of tutorial, so don’t hesitate to add a feedback.

Converting Lop Ears with Arkos Tracker 3

https://youtube.com/watch?v=pY0fdEDKpok

Let's adapt Lop Ears (by Andy Severn) with Arkos Tracker 3!

Creating drums

Drums are notoriously hard to create, and there is no definite guide… And this page isn't one, sorry :). However, we will see through various examples how to build decent ones.

First advice

One thing that is important to remember is: **one drum may sound good for one song, but not for another one**. For this reason, I would always advise, like for all sounds, to create your drums from scratch for every new song. Even if you have a "formula" that works for you, apply it, but from scratch. By not copy/pasting/loading, you will introduce small variations, maybe errors, and the song will benefit from it.

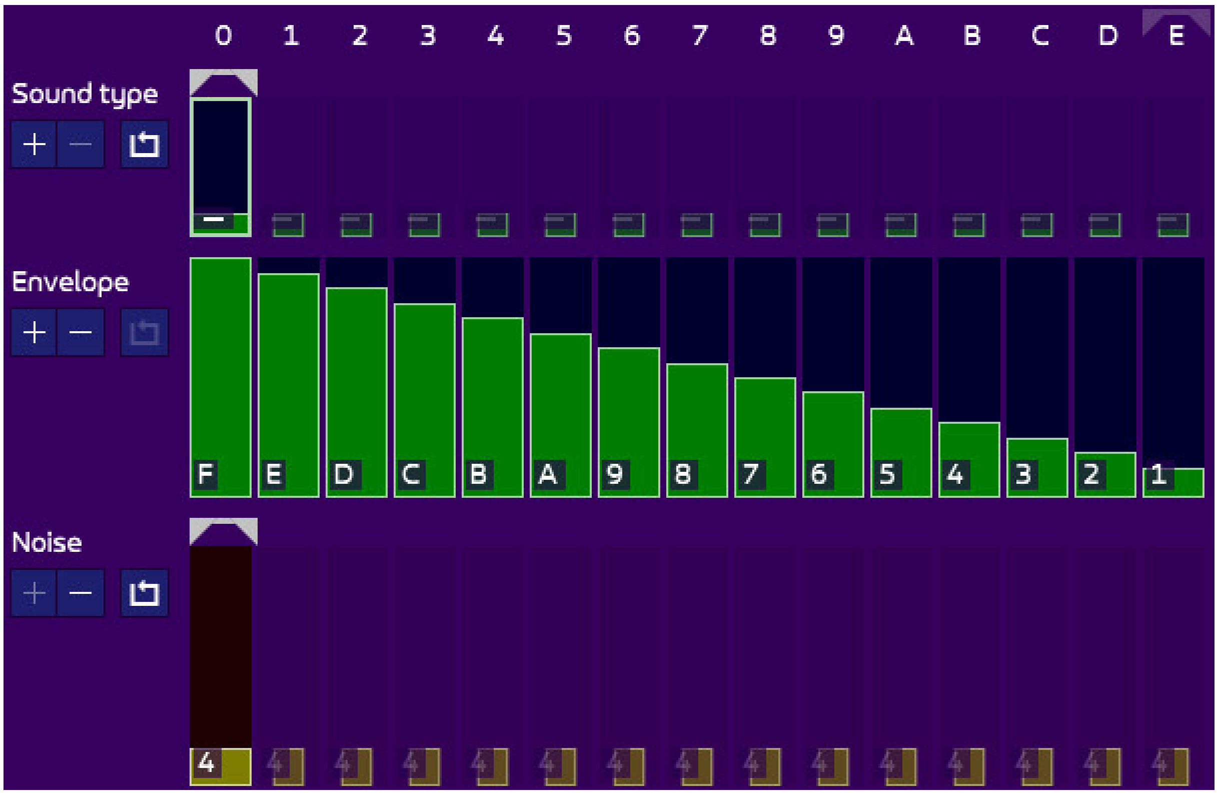
It is the mix of all your sounds that make your song stand out. One benefit of AT is the possibility to play a pattern while editing your sounds. So don't hesitate to play one of them, all the better if it includes even basic arrangements, like a bass, basic drums, chords and melody. Then enter the Instrument Editor and start tweaking the drums in real-time.

The funny thing is that, the sounds of songs you love often sound thin, even crap, when listened to alone, as we'll see in the next examples. Hence the need to listen to them along your arrangements!

The basics

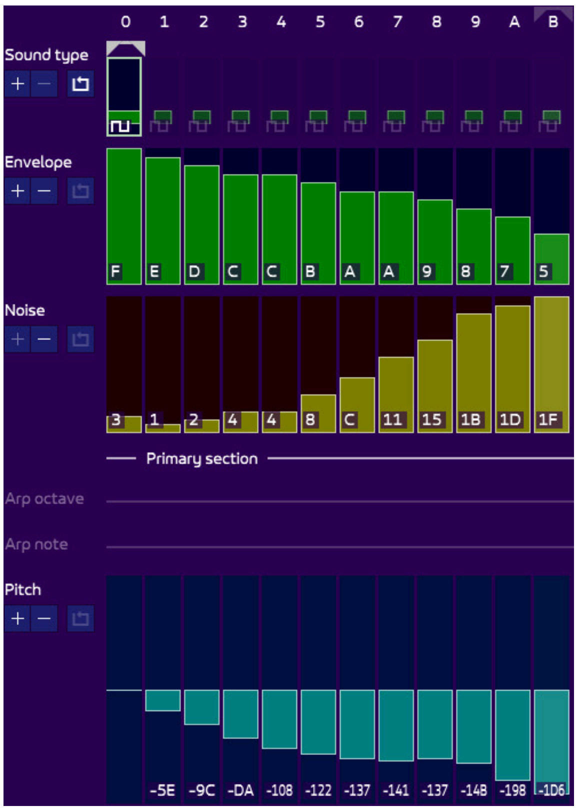
Drums are generally made of two things: noise *and/or* square envelope, with a decreasing pitch (I personally have never heard hardware envelope successfully used in a drum, but I'm sure there are some out there!).

A lot of old-old-school drums were simply composed of noise :



0:00 / 0:00

Simple for a snare, but probably too simple. So let’s add a square wave, with a decreasing pitch. I also made the noise period evolve for good measure. It’s all about experimenting!



0:00 / 0:00

Much better, isn’t it?

Notice the *and/or* I mentioned before. The trick to most “evolved” drums is to alternate the square wave being present, probably without noise, and no square wave but with noise, the latter being used at the beginning to produce an attack. Examples below!

Real-world examples

It’s time to use the world’s best trick to rip sounds from music: the Streamed music analyzer (SMA)! It’s a tool from AT to play and extract sounds from music exported into the YM or VGM formats. It’s all explained here, so I won’t do it here again, but know that the examples below were created using this tool.

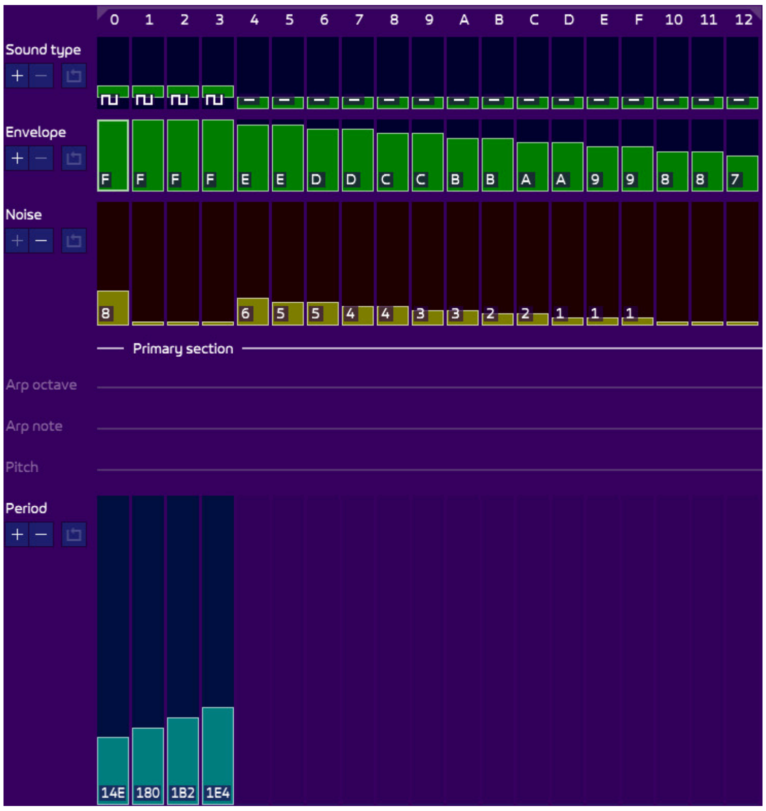
The following examples, used in many great songs, show many interesting things:

- They don’t sound as good as in the song itself!
- They all use more or less the same pattern.
- They are a trademark of their author.

Note: all these examples are in the package, in the Instrument folder.

Count Zero snare

Let’s begin with the snare of Count Zero:

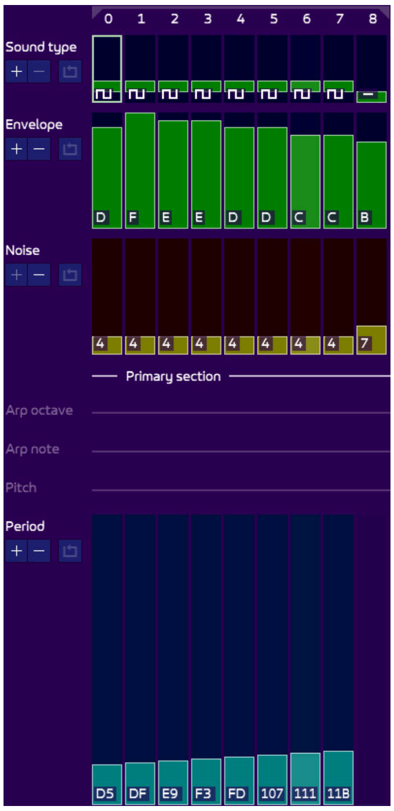


0:00 / 0:00

It is interesting because it starts with the software period without noise, except the first frame where noise is present for the “attack”, and ends with noise only. It is pretty original and sounds powerful on its own!

Tim Follin snare

The snare dum used by Tim Follin in Led Storm goes against what I said, as it is only a noise a descending square envelope:



	0	1	2	3	4	5	6	7	8
Envelope	D	F	E	E	D	D	C	C	B
Noise	4	4	4	4	4	4	4	4	7
Period	D5	DF	E9	F3	FD	107	111	11B	

0:00 / 0:00

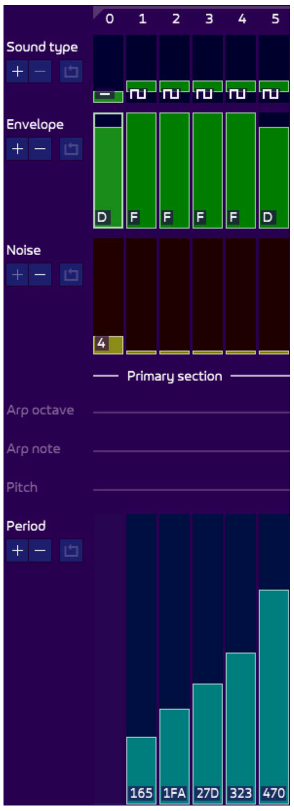
Funny to notice that the first frame has not a full volume, set only on the second. Yet it works great, especially in these awesome tunes.

Bassdrums

Let’s now examine some bass drums. A lot of them consists in an attack with noise, then a descending pitch. Here are three of them from famous composers. Can you guess who’s who?

0:00 / 0:01

Answer: Big Alec, Lap, Madmax! The one from Lap looks like this:



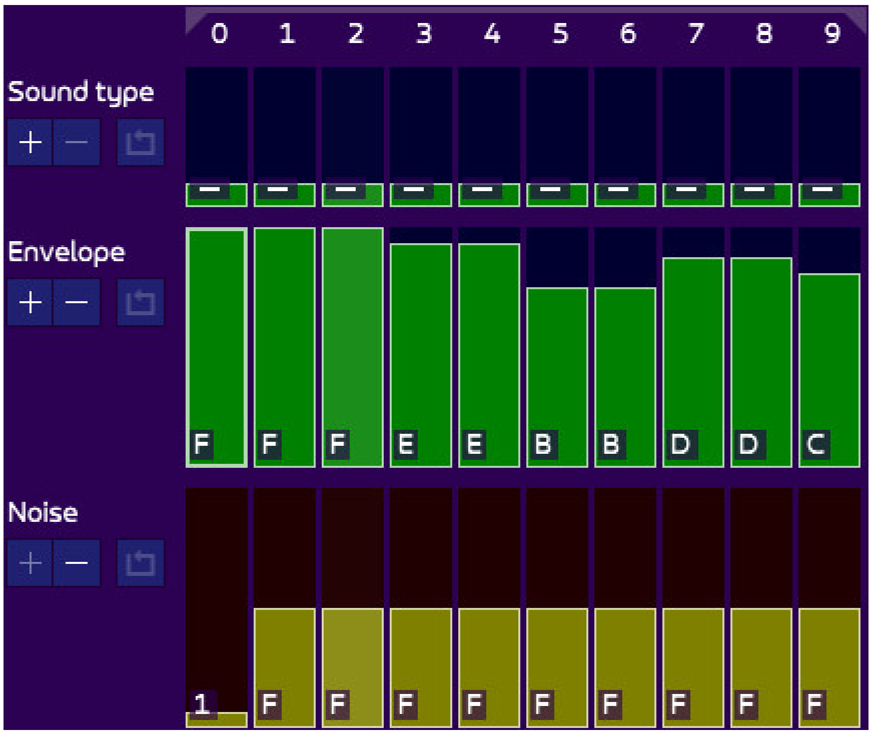
The other two use the same formula. All three have a lower volume on the first frame! Is it a trademark for Atari ST musicians??

David Whittaker snares

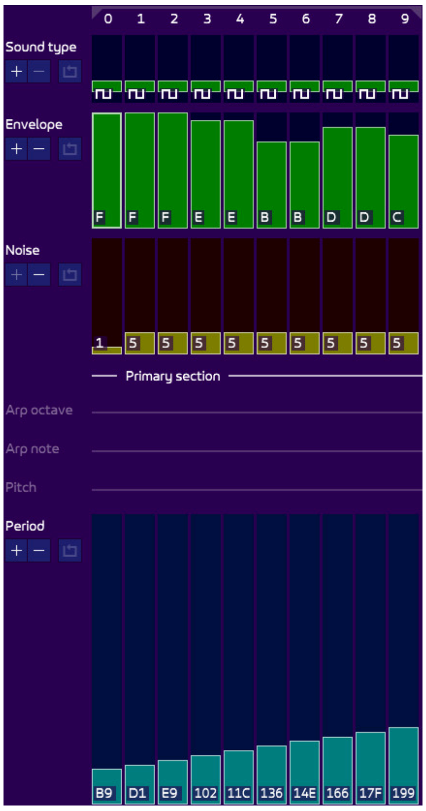
The mighty David Whittaker rarely (…never?) uses bass drums, but has two snares played alternately, which is his trademark:

0:00 / 0:01

The first one is raw noise with a treble in the volume:



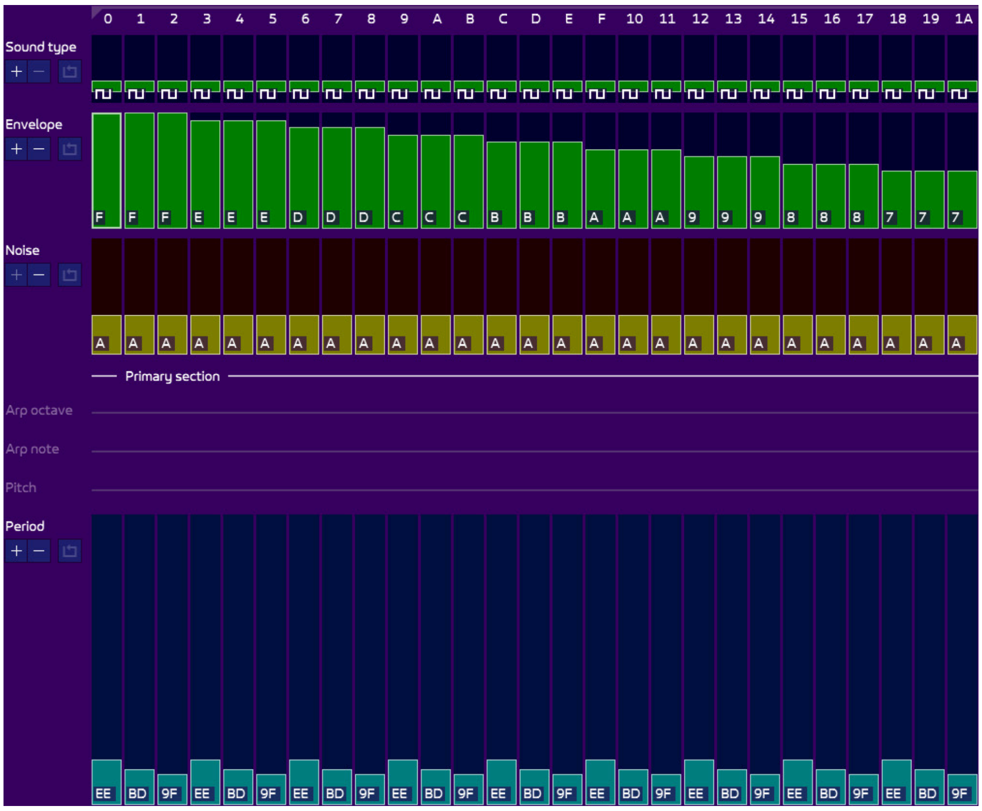
The second adds a descending pitch with a higher-pitched noise:



Chris Mad snare

Special mention to Chris Mad for his very original snare drum:

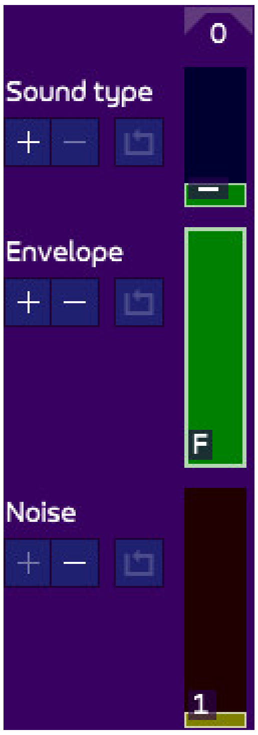
0:00 / 0:01



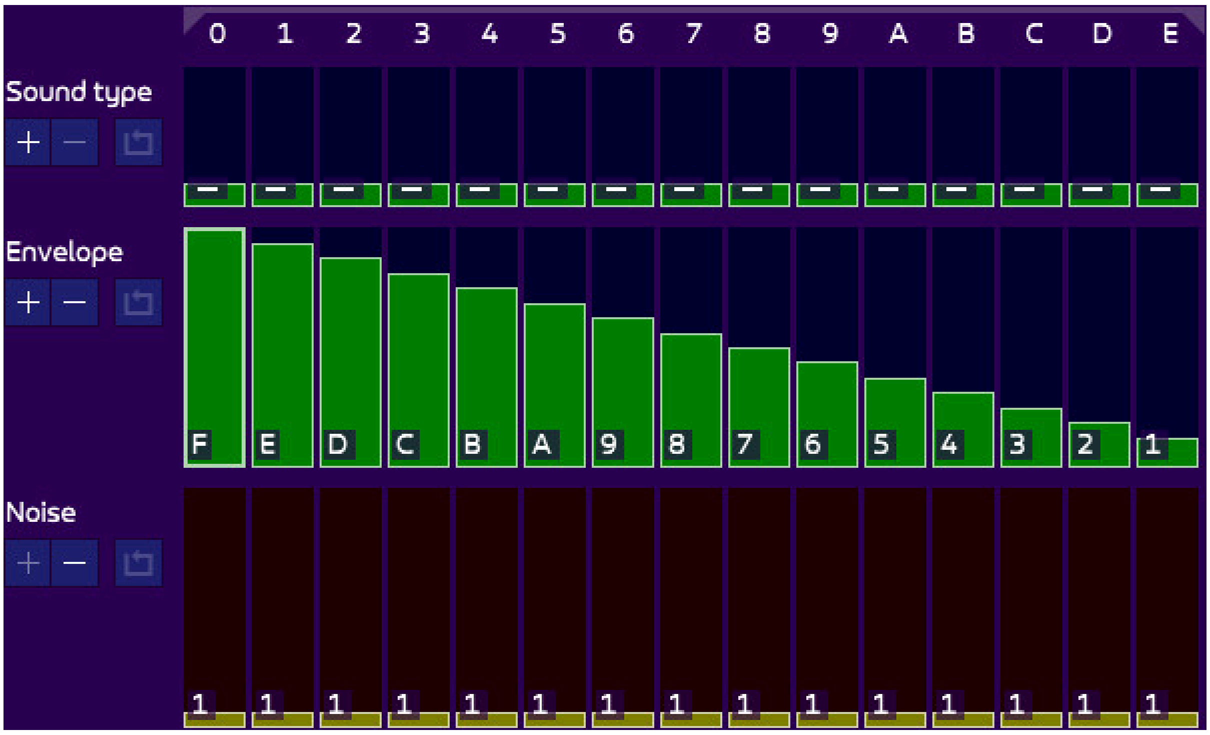
It is actually an arpeggio along with raw noise. Which allows to be both a percussive and melodic element of the song! To expand this concept, one would not force the period, and simply add an arpeggio, either in the instrument or, for more versatility, in the track itself.

Hi-hats

Hi-hats can be done very simply with a single frame of noise, without square envelope most of the time:



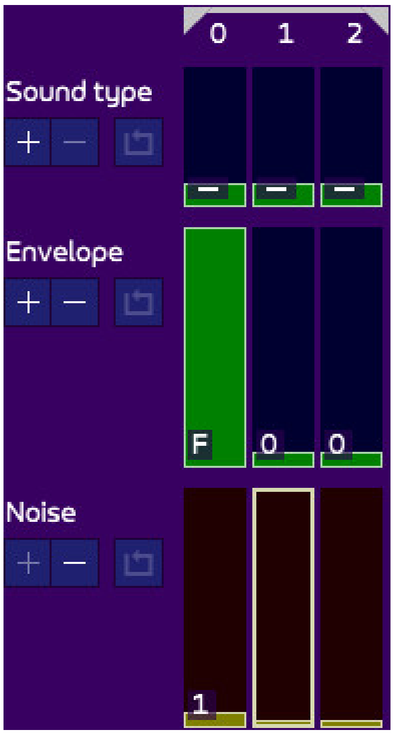
You can create an open hi-hat by using a decreasing volume envelope:



Used together, it could sound like this:

0:00 / 0:01

A nice trick is to create a looping sound timed to twice your tempo to simulate a double hi-hats, without having to change the tempo of your song:



Note the sound loops on three frames, so our speed must be a multiple of that, else it probably won't sound right. It sounds like this when used in a song (with a speed of 6):

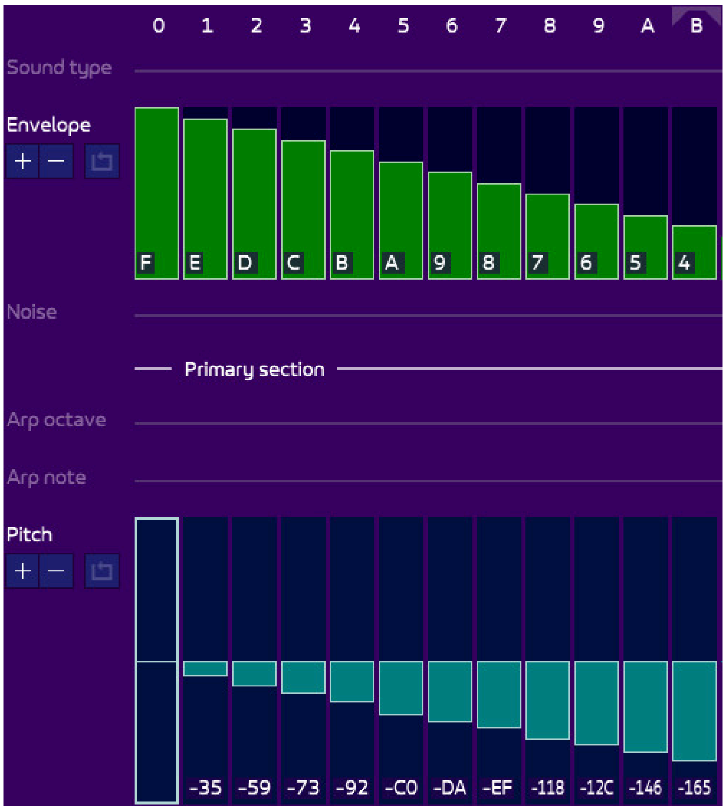
0:00 / 0:01

Another trick is to add a single frame of noise into an instrument, like a bass or a chord, adding a percussive element to it:

0:00 / 0:01

Toms

Toms are usually a simple sliding-down pitch, without noise. This time, don't hard-code the period: use a pitch so that you can have several toms with just one instrument:



.... which, used in a track, could be used like this:

0:00 / 0:01

Wrapping up

As we have seen, there are ways to create generic and efficient drums, but drums are actually very personal instruments, and one can recognize the author by simply listening to them! All the more important to create your own style, but that goes only by experimenting. But don’t be afraid to study the sound of others to create your own!

Mastering hardware sounds

“Hardware sound” is the generic name given to the envelope generator. It is probably the most used of the “advanced” tricks of the AY/YM. However, mastering it is not that easy, especially if you’re not aware of the high limitations it has. You *should* read the first tutorial of this website about understanding the PSG beforehand, but since you’re here (and I’m a nice person), here’s a little summary, plus a little more:

- There is only **one** hardware envelope generator. This is the most important of all these bullet points.
- The generator can be used on **any** channel**s**, i.e. one or more.
- If using the generator on more than one channel, all these channels will use the same hardware envelope and hardware period (the software period remains their own).
- The generator is only in charge of changing the volume of the channel(s), according to its own period and shapes (see below).
- When using the generator, the volume of the channel is managed only by the generator, you cannot set a volume on your own anymore.
- There are 8 different shapes (or curves), looping, looping in reverse, or staling. But they always go from 0 to 15 or the opposite, which is probably the biggest limitation. There are no build-in ADSR curves like on more advanced PSGs, and you cannot make a curve go from 5 to 10. No, no, no.
- Once the envelope starts, it can be started again at any time (technically, by feeding the PSG with the same curve number again). It is what we call *retrig*.
- The goal of the PSG engineer was probably to relieve the developer from changing the volume by himself. Sadly, the generator is so limited there is no incentive to use it as-is in a song. Unless…
    - You are involved in a size-coding production.
    - You want to trigger a long decaying hardware sound while loading a file! As Zik did in the great Ghost Nop demo…
- Software periods are encoded on 12 bits, hardware periods on 16, so they are technically more accurate.
- Remember the PSG deals with *periods*, not frequencies. The higher the frequency, the lower the period. This has some very real consequences: the period is just a decreasing integer counter: when it reaches 0, the PSG starts a new cycle, hopefully producing a sound at the right frequency. But what if the counter is not accurate enough, and the real frequency lies between period 45 and 46? Well, no luck here, the sound will be out of key. Faster PSG (such as the YM

running at 2 Mhz) will have less problems than a slow CPC PSG (1 Mhz), but it will have problems anyway.

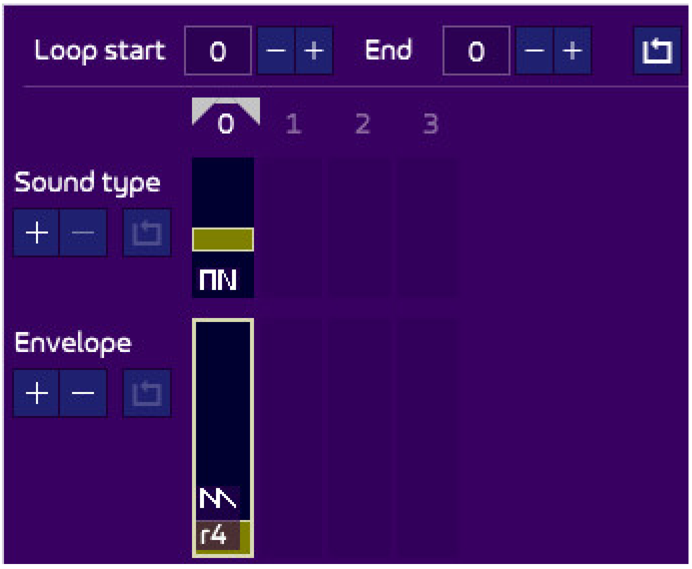
So now you've understood it, the envelope generator is pretty limited, but its real strength resides in tricking it into something it wasn't designed for.

As seen in the tutorial mentioned above, by synchronizing the period of a square generator and the envelope generator (typically with a ratio of 2^x, with x = 4 or 5 being the most common. That is, the hardware period is 8 or 16 times faster than the software one), we get a square sound that is modulated by another waveform, creating a whole new sound:

0:00 / 0:11

Simple sounds for starters

Let's start the journey with the simplest, classical, yet very nice, "curve 8 (descending sawtooth), ratio 4". In the Instrument Editor (IE), it would look like this:



There is only one looping column. The sound type indicates "software to hardware" (square to hardware envelope), meaning that the primary period (the software period) comes from the music score (i.e. the notes of your music), and the secondary period (the hardware period) is calculated from it. Other sound types are explained below. The envelope, as you can see by the little drawing, is the descending sawtooth, using a ratio of 4.

And by going from octave 1 to 5, it would sound like this:

0:00 / 0:11

Inaccuracy?

The four first notes sound great, but awww, the fifth really bad. Why? Because the higher the note, the smaller the period, so the bigger the possible inaccuracy. You hear the “mwwwaaaaaah” sound that we like so much? Well, it happens *because* of the inaccuracy! So a bit of it is ok, but too much is *meh*!

Let’s dive a bit into numbers. The software period is calculated in real-time, but you can find it in your CPC manual too (what a great book it was!).

The hardware period is the software period divided by 2^4 (=16), as stated before, because we use a ratio of 4.

Note	Software period (1 Mhz PSG)	Hardware period, theoretical	Hardware period (1 Mhz PSG)	Error
C-1	1911	119.4375	119	0.4375
C-2	956	59.75	60	0.25
C-3	478	29.875	30	0.125
C-4	239	14.9375	15	0.0625
C-5	119	7.4375	7	0.4375

The player doesn’t use decimal numbers, because the PSG doesn’t, so it rounds to the higher integer. As you can see, nothing is perfect and the error value goes high or down according to the notes. The funny thing is that the error value for C-1 and C-5 is the same, but C-1 sounds good and C-5 does not. Why? Because the period of C-1 is higher, so there is less room in the signal for the error to be heard.

Interestingly, here is the same sound with a 2 Mhz PSG, so twice faster than that of the CPC, like you would find on Atari ST (though they have a *YM*, not an AY!):

0:00 / 0:13

It sounds mostly much more accurate (less “mwwwwwwaaah”), especially the last note. And the periods look like this:

Note	Software period (2 Mhz PSG)	Hardware period, theoretical	Hardware period (2 Mhz PSG)	Error
C-1	3822	238.875	239	0.125
C-2	1911	119.4375	119	0.4375
C-3	956	59.75	60	0.25
C-4	478	29.875	30	0.125
C-5	239	14.9375	15	0.0625

Interestingly, the notes with most errors aren’t the same as with the CPC. And their software period is much higher, making the error less noticeable.

Did you notice that for the same note (C-1 for example), the periods are *doubled*! Since the ST YM is more accurate, you can imagine that the range of the periods are *stretched* across the 12 bits of the software period, thus giving more accuracy.

Solutions against inaccuracy?

Is there something we can do against it? Well, yes and no.

- The CPC is the slowest of the pack: its AY runs at 1 Mhz, so the periods must be smaller to produce higher-pitched sounds: calculation with small numbers *will* provoke miscalculations. So one solution is to change your hardware! This might not be the solution you’re asking for…
- You can change the *frequency reference* in your subsong properties. This is not something we usually do (on CPC), but that I’ve seen on Spectrum… and in some Dave Rogers’ songs. By changing the frequency of the reference note (C-4), from 440 to another one, you might find that the error is shifted and it might be useful to *your* song. It also involves using another period table, which takes more memory if you need several specific tables for various songs. Also, people with the perfect pitch ability will cringe, but hey…
- By decreasing the ratio, you have less inaccuracy. But it won’t sound the same… Sometimes you will use one ratio for a specific note to counter an off-key sound. The listener might not notice.
- Adding a vibrato or some effect might *hide* the inaccuracy, as we’ll see below.

Other basic sounds

The most basic thing you can do is simply change the waveform and the ratio:

- The two sawtooth curves are more “aggressive”.
- The two triangle curves sound smooth and are perfect for more ambient sounds.

Which to choose between two curves of the same type? There is no difference when using them stand-alone, so they are interchangeable most of the time.

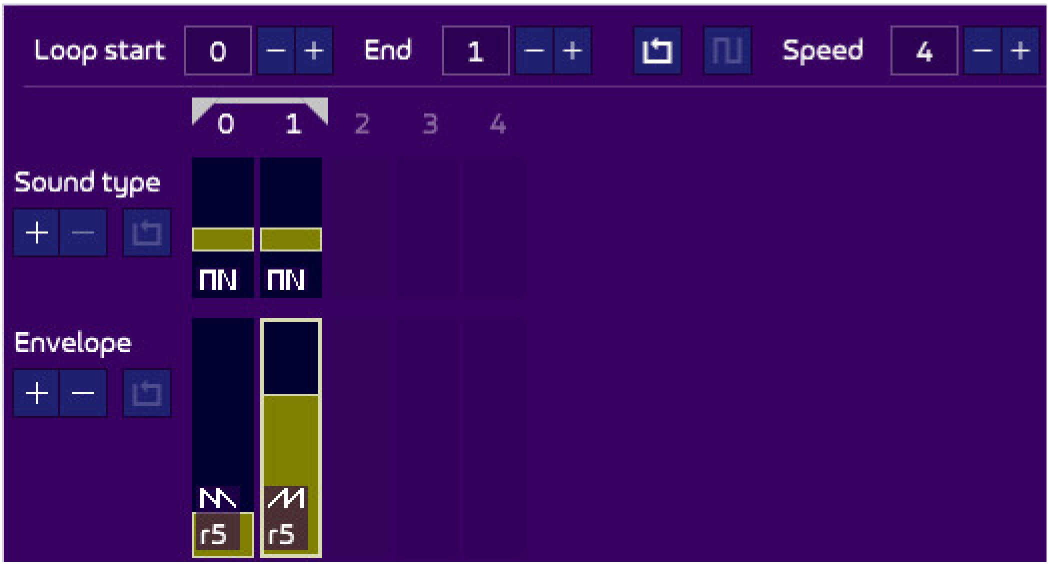
For now, here is a different bass line using, with a ratio of 5, first the sawtooth curve, then the triangle curve. Notice how the second is less aggressive:

0:00 / 0:15

Alternating curves and ratio

Creating original sounds isn’t complicated, and one way of doing it is by changing quickly the curve and/or the ratio. Here is an interesting answer to “what two curves of the same type are for”?

0:00 / 0:04


Loop start 0 End 1 Speed 4

The speed has to be set to 4, less would sound to harsh, but it really depends on what sound you want.

Sound types

So far, and in most hardware sound examples, we've used the sound type "software to hardware", because it's the most typical and, well, it works fine 99% of the time!

But let's delve a little more and explore the other possibilities. First, you should read the sound types and what they are up to.

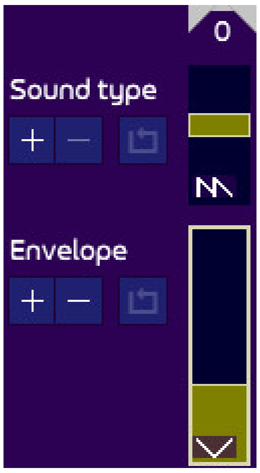
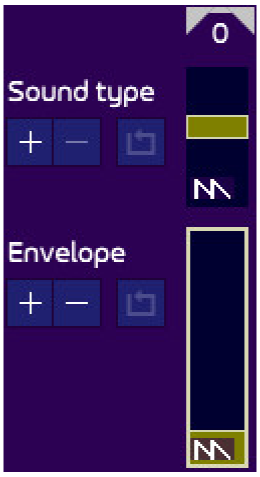
Hardware-only sound type

This is not an often used sound type, but it has a very unique personality. When used, the software square wave is not produced at all. So only the hardware envelope is generated. This has two major consequences:

- It sounds much softer than the classical soft-to-hard, which is why it can be overwhelmed if used along loud instruments.
- Since there are no other signal to be coupled with, it cannot be desynced. So it is a stable, *mwaaaah*-less sound.

0:00 / 0:08

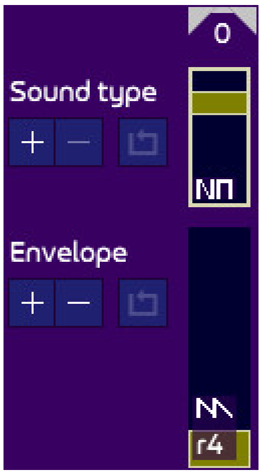
The two sounds use the shape triangle and sawtooth, respectively:

But this sound type may not be suited to high-pitched sounds, as inaccuracy strikes it badly.

Hardware-to-software sound type

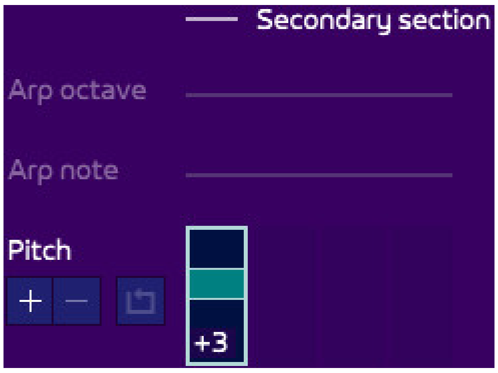
This sound type is like software-to-hardware, but the hardware period is calculated first from the music score, then the software period is multiplied by the desired ratio. As a multiplication is used, there is no rounding, so no inaccuracy, so no *mwwwwaaaaahh* sound. Just like hardware-only, it is a perfectly stable sound, but with both the software and hardware sound.



0:00 / 0:07

Going into higher-pitched sound will sound off-key. Why, you think, as you thought the hardware period was more accurate? Yes indeed, but by multiply it with a ratio, we may actually miss the right software period. If we take the C-1 note from the table above as an example, the hardware period is 119, which we multiply by 16, giving 1904. Ah! But the software period should be 1911! So there, we are off-key.

Also, the sound may feel a bit flat as-is, so what about adding our very own controlled *mwwaaahhh*? It is simple: add a pitch value in the secondary section. It will be added to the software period! Indeed, in a hardware-to-software sound type, the primary section relates to the hardware, the second to the software:



0:00 / 0:07

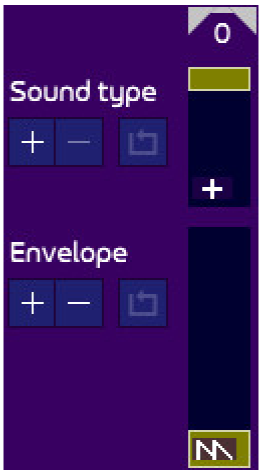
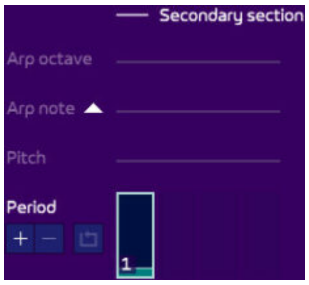
Software-to-hardware sound type

Last but not least, this sound type is for rather specific needs. Both period calculations are independent, so you can for example, force a hardware period, or add an arpeggio to the hardware period only. It allows you to go beyond the “ratio rule” that was performed in the soft-to-hard and hard-to-soft sound types.

One first example is indeed forcing the hardware period, which Ben Daglish was famous for in most of his songs.

0:00 / 0:03

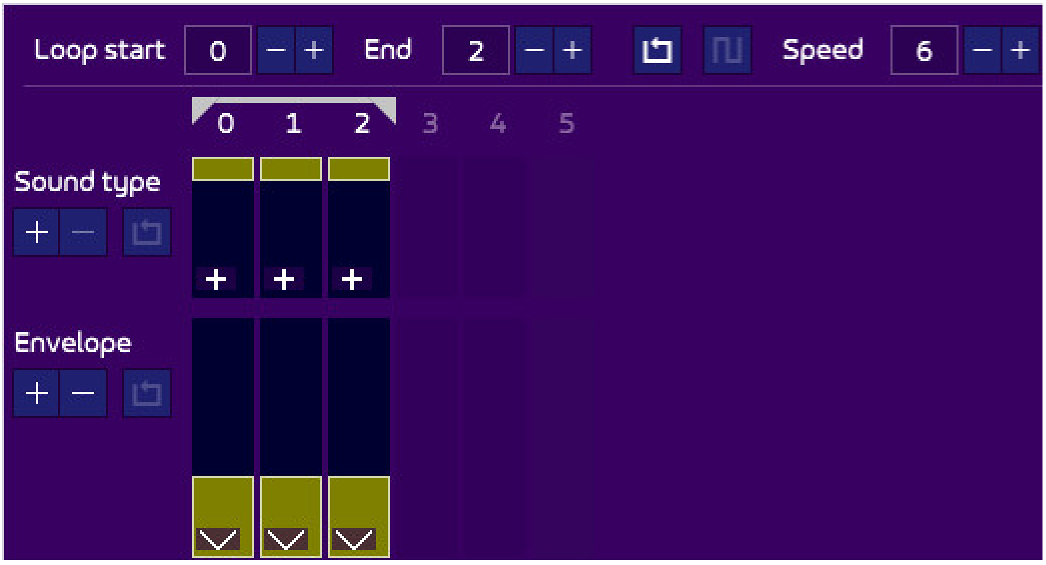
The first part is done by forcing the hardware period to 1, the second to 2:

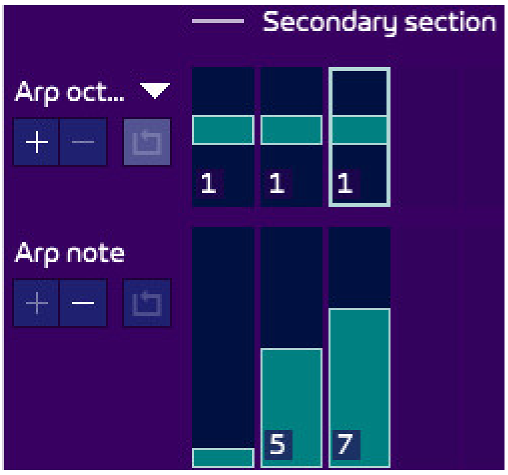
	

Another unique possibility with this sound type is setting one of the period to its own, via a arpeggio, thus creating some kind of polyphony:

0:00 / 0:05

Such sound is built like this:



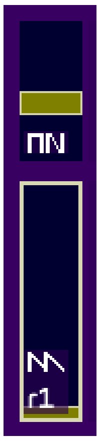


This is not the sound you hear every day!

Hardware melody

Musicians have a tendency to use the hardware envelope only for bass, because, as we've seen in our example earlier, high frequencies don't sound good. That's not always the case!

The problem is the ratio that is too high. But if you set it to a lower value, you can do great stuff. Here is a raw example using the low ratio of 1, sawtooth, starting at C-0, going octaves higher, and playing a melody:



0:00 / 0:15

Some notes don’t sound right, so you may be limited, but there is one trick to correct that: using a vibrato to “hide” the inaccuracy. Read on…

Add some effects!

Many musicians simply don’t think about adding effects to the hardware bass. There are some reasons why:

- They lack imagination.
- Their software simply doesn’t allow changing the hardware period every tick.
- Their software doesn’t automatically calculate the hardware period from the software one, so adding a vibrato, for example, wouldn’t sound good.

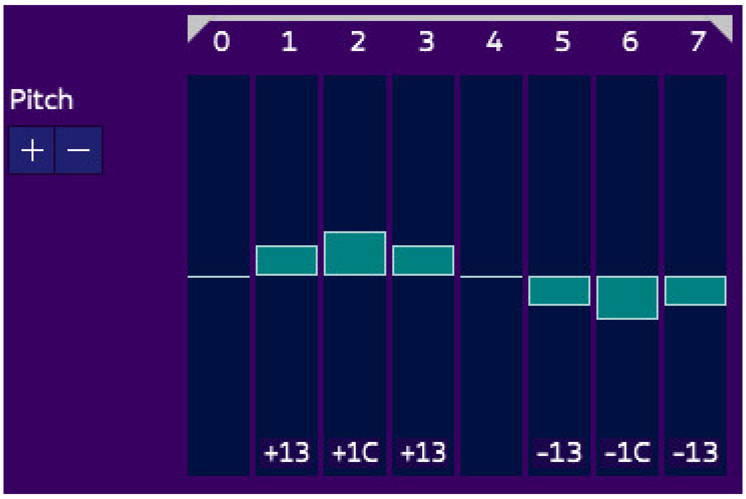
A good thing such musicians have now discovered AT 🙂 !

Adding vibrato

Let’s add a vibrato to a hardware bass:

0:00 / 0:10

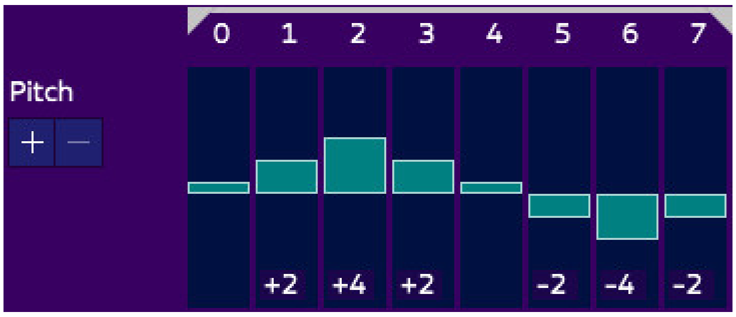
Not bad, he? The tricky part is to get the amount of vibrato right. Here is the pitch table I used:



As you can see, it uses quite large values. If I'd used smaller ones, you wouldn't have heard the vibrato. However, such pitch will not sound right for high-pitched sound. So you will probably have to create several vibratos according to the frequency of your notes. The higher the pitch, the smaller the vibrato frequency range.

Here is a hardware melody with a small vibrato (you *don't want* to hear it with the large vibrato above!):

0:00 / 0:10

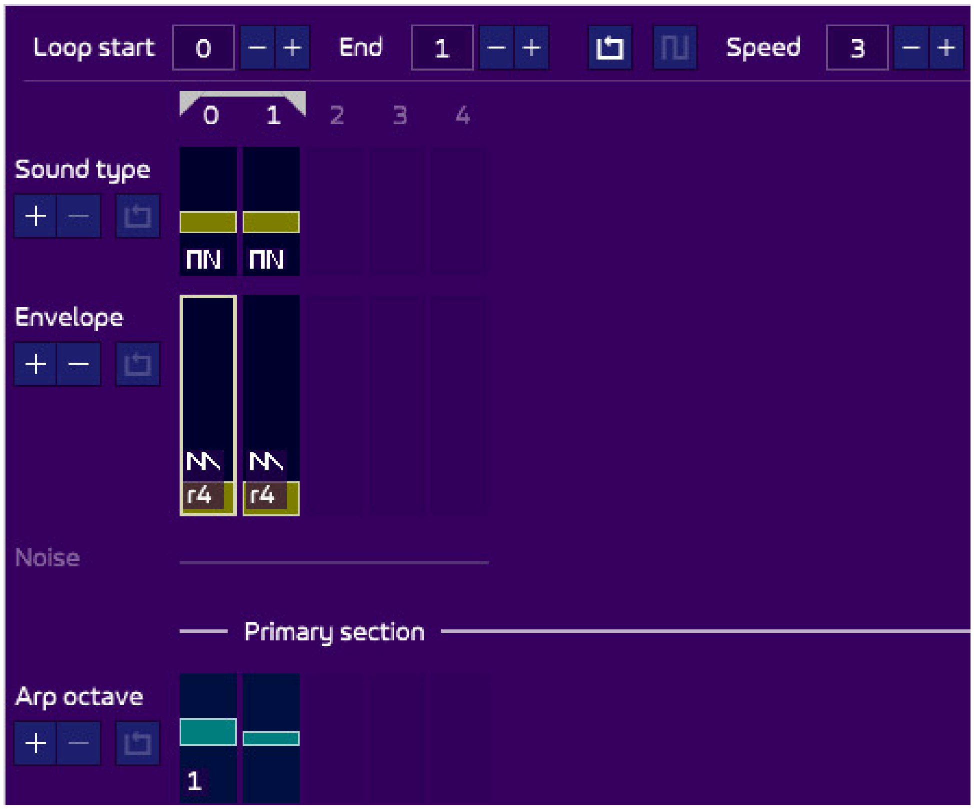


On top of sounding rather original, the vibrato helps *hide* the lack of accuracy of the periods, so it's a good trick to play melodies without them sounding too *cringy*.

Adding arpeggio

One other effect that I almost never heard (besides in my own songs) and that we presented in an example above, is an arpeggio coupled with hardware sounds. It is so effective!

0:00 / 0:07

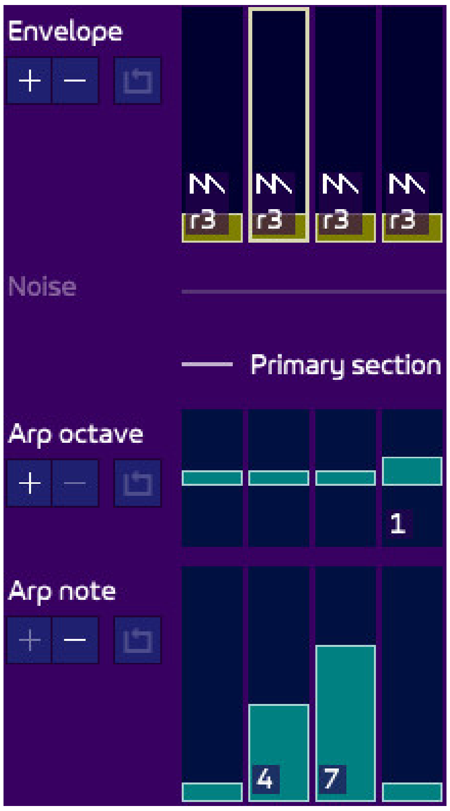


Two things to notice:

- Most of the times, such sounds will not sound right at full speed. Here, I used a speed of 3, though faster speeds may give yet another special effect.
- I suggest that such arpeggio should however be encoded within the instrument itself, not besides using an arpeggio table. Why? Because in order to sound good, the ratio often must be adapted to the octave/semitone change, and it's much easier to tweak that in one place.

Here is another example using a full chord:

0:00 / 0:07

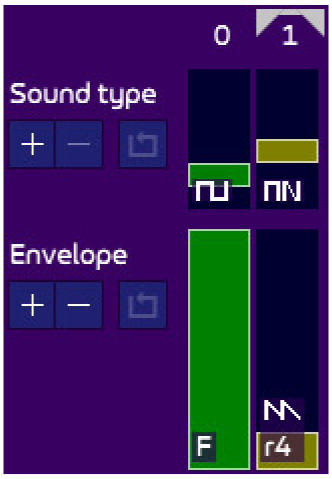


Attack attack!

One thing you may have noticed while playing with hardware sound is that it might lack a bit of "attack". Sure, the generated volume keeps changing, but the average volume is the same, so the start of the sound may not be as noticeable as if you'd created a simple software sound with the good old 15-14-13-13-13 curve. There are at least two tricks to it.

Attack with another type of sound

AT gives you the possibility of starting your sound with something, but not loop on it afterwards. One thing I often do is start my hardware sound with a software frame:



0:00 / 0:05

Notice that the sound loops only on the second column. Depending on your sound, you may need to make an octave jump on the software period, but that's up to you to test.

Retrig

As said at the top of this page, the hardware envelope can be restarted at any time. This is called *retrig*, and AT gives you two ways of doing it. Both are explained here, so I won't write the explanations again, but here is how it sounds like:

0:00 / 0:06

You can also add retrig on any column, such as the chord we saw earlier, and see what happens. It might sound good… or bad!

Hardware sound on multiple channels

This is a lengthy subject which I will only scratch here, as many experimentation can still be done.

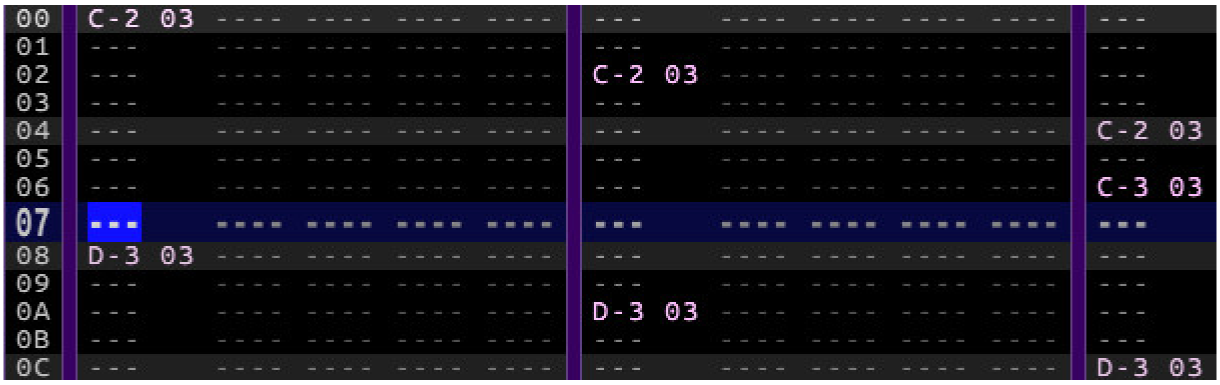
As we've seen, there is only one hardware generator, but it can be applied on as many channels as you want. The hardware period and envelope will be the same on every channel. If the software periods are synchronized, all will be fine. If not, it will probably sound bad, yet some musicians managed to produce interesting things with that.

Priority

AT manages its channels from the first to the last (I'm talking about the index, not the stereo positioning. Some PSGs have their channel 1 on the right). It means that, in the user interface, the track on the left has less priority than the one on the right. So if one sound in the channel 1 uses a hardware sound, and the channel 2 another one, they will both use the hardware period and envelope of the channel 2. This notion is important if you want to mess with using several hardware sounds.

Several channels test

Let's try to have several hardware sounds! We'll start with something simple:



```
00 | C-2 03 ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---
01 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---
02 | ---    ---- ---- ---- ---- | C-2 03 ---- ---- ---- ---- | ---
03 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---
04 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | C-2 03
05 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---
06 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | C-3 03
07 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---
08 | D-3 03 ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---
09 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---
0A | ---    ---- ---- ---- ---- | D-3 03 ---- ---- ---- ---- | ---
0B | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---
0C | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | D-3 03
```

0:00 / 0:13

From line 0 to 4, we add the same hardware sound to all channels. This sounds right, because the notes are the same, so both software and hardware periods are in sync.

On line 6, the leading channel (the third one, with the highest priority) goes to octave 3: it still sounds right because the hardware period, though doubled too, is still a multiple of the software periods of all the channels.

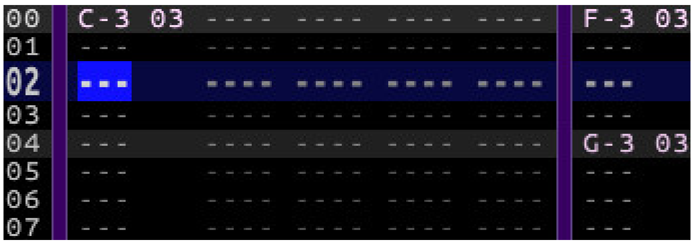
Trouble comes on line 8: the first channel uses another software period, but the hardware period doesn't change accordingly: the third channel has priority and won't accept that. So the channel 1 sounds like crap.

Same thing on line A: the second channel gets in trouble too.

Then the miracle happens on line C: the third channel uses the same software/hardware period as the two other channels, which are now in sync.

Harmony through desync

There are a few cases where channels can naturally sound right with multiple hardware sounds:



```
00 | C-3 03 ---- ---- ---- ---- | F-3 03
01 | ---    ---- ---- ---- ---- | ---
02 | ---    ---- ---- ---- ---- | ---
03 | ---    ---- ---- ---- ---- | ---
04 | ---    ---- ---- ---- ---- | G-3 03
05 | ---    ---- ---- ---- ---- | ---
06 | ---    ---- ---- ---- ---- | ---
07 | ---    ---- ---- ---- ---- | ---
```

0:00 / 0:06

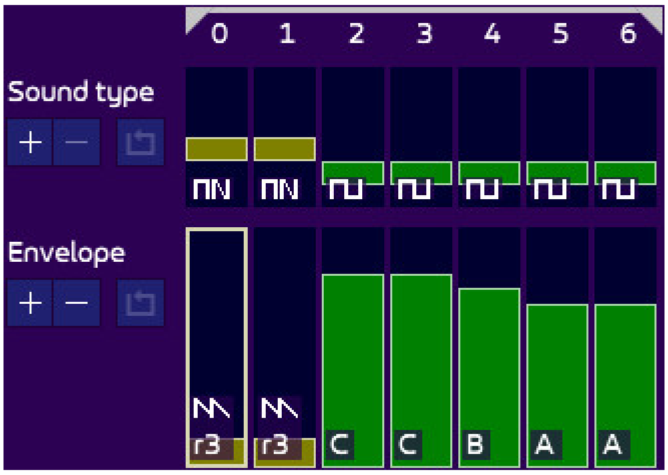
Sounds nice, eh? Such “miracles” can be found by testing, or probably by calculation, which I’m not interested in doing (!). This kind of relates to what we did before with one channel only, using the sound type “software and hardware”.

Alternating channels

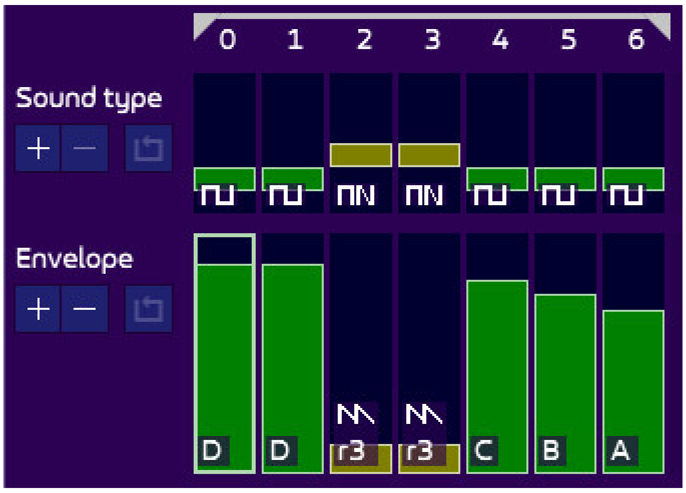
One trick I also used but actually never saw anywhere (should I patent it?), is to quickly alternate the hardware between the three channels. Only one at a time will use it, so there will be no desync, but it still adds an original sound to the music. Here is an example from the menu music I did for Axelay’s Star Sabre 128k (which you can find in the AT package, in the STarKos folder):

0:00 / 0:13

The lead sound looks like this on channel 1…



… and on channel 3:



And the bass on channel 2 produces its hardware bass on ticks 4 and 5. No clash, and it sounds original and great!

Bonus

As a bonus of what can be done through experimentation, here a *four-channel* sound using three channels of hardware sound, from Sparklite's Amstrad Export 2005 cheat part demo:

0:00 / 0:07

It may not be the typical sound you'd use in a song, but it sure proves that many things can still be done!

Wrapping up

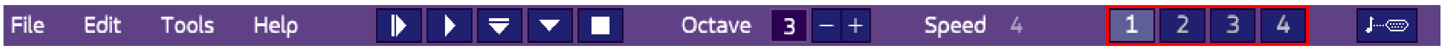
This is the end of this tutorial, which, I think covers most of what can be done with AT! Now go experiment, have fun, and create great music!

The layouts

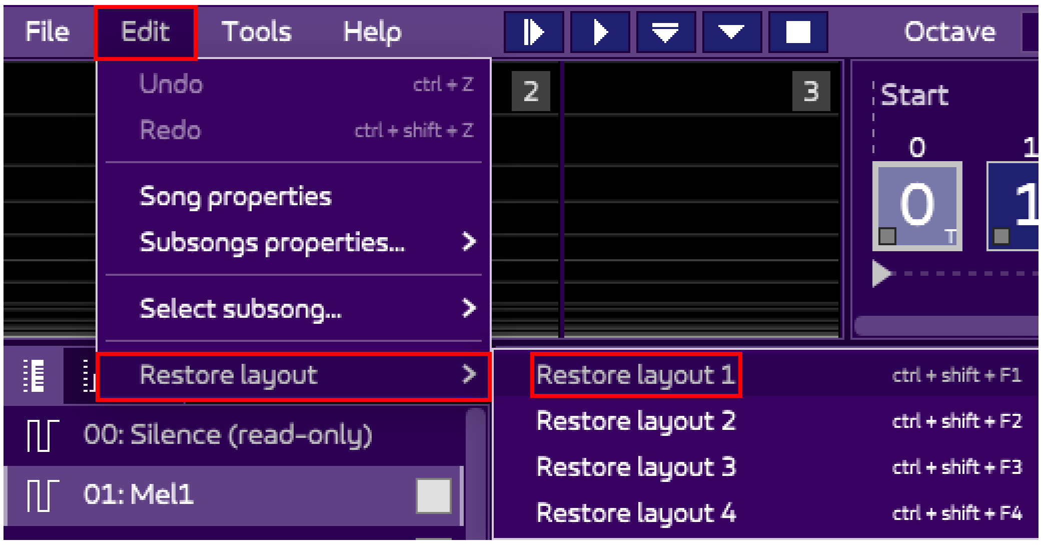
The user interface in AT3 is flexible: several *layouts* are available for you to focus on particular task, or on the contrary, to have a view on as many elements as possible. For example, one layout has the Pattern Viewer filling most of the screen, another one shows the expression lists and both their editor.

Layouts can be selected in three ways:

- By the layout buttons at the top of the screen:



- By using the menu at the top left:



- Or via a keyboard shortcut: ctrl + shift + F1 up to ctrl + shift + F4

Selecting a layout automatically changes all the panels of the screen.

Layout 1 is the default layouts and shows a bit of everything.

Layout 2 focuses on the Pattern Viewer or the Instrument Editor.

Layout 3 shows the expression lists and their editor.

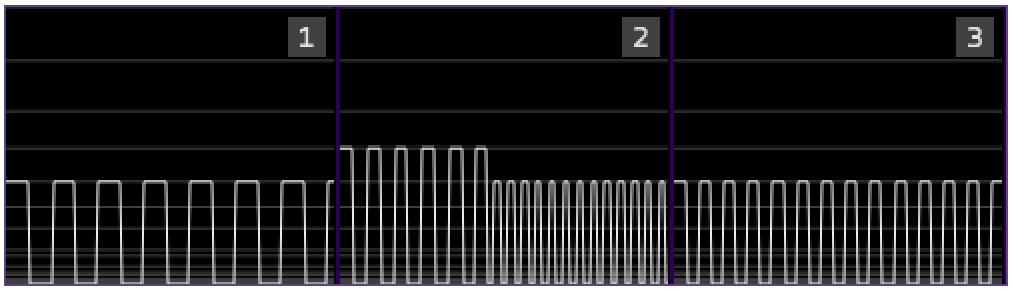
Layout 4 is more for a “listening” experience, with only the Linker, Meters and Pattern Viewer.

When inside a layout, you can use tab or shift + tab to switch to the next/previous tab.

Need more layouts? Contact me and I may add more!

The meters

At the top-left are the meters:



There are as many meters as the PSG used by the song has channels (3 for most 8-bits, or more if using specific hardware such as PlayCity, Darky, SpecNext, etc.).

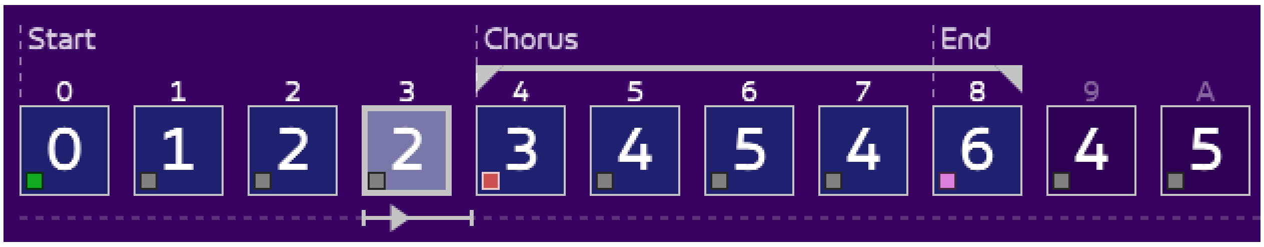
The meters show the waveform of each channel.

The only interactions are:

- Left-click to mute/unmute a channel.
- Right-click to solo/unsolo a channel.

The linker

At the top of the screen lies the Linker (**LK**). It is basically a sequencer of patterns, which contains tracks, which you must have heard about if you have read this tutorial. If the linker of a new song is rather empty and unexciting, here is a more extensive example:



Read this from left to right.

Positions and patterns

Each “box” is assigned a number at the top: 0, 1, 2, 3, up to A (hex) in this example. This is the **position number**, which is actually only an increasing number. So, to a position is assigned a “box”.

Inside each box is a big number: this is the **pattern number**. As you can see, patterns can appear several times, which is useful to repeat the same part of the songs without having to duplicate its content manually.

The selection

On position 3, you can see the box being highlighted: this is a selection, which you can define by clicking on a position with the mouse, or with the following keys:

- cursor left cursor right to move backward/forward.
- shift + cursor left shift + cursor right to extend/shrink the selection.
- page down page up to move backward/forward faster.
- start end to move to the beginning/end of the song.

You can also perform the following operations:

- right-click to open a contextual menu.
- space to edit the position (see below).

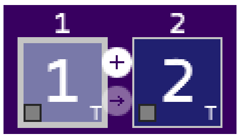
- ctrl + shift + up/down to increase/decrease the pattern number (within the existing ones).

Creating/duplication/cloning

Various operations can be done on the positions, via the keyboard…

- ctrl + n to create a new position with a new empty pattern.
- insert to duplicate the position (the same pattern is used).
- shift + insert to clone the position (a new but identical pattern is created).
- delete to delete the position.
- enter to open this position in the pattern viewer, focusing on it.

… or via the mouse, by hovering above a position:

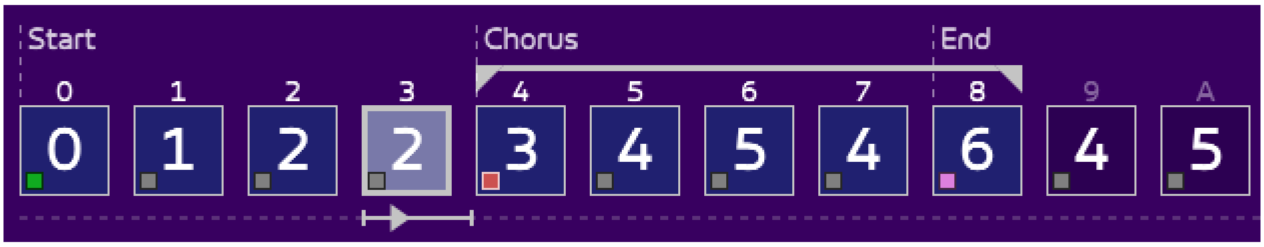


The "+" icon will duplicate the position, the arrow at the bottom will clone the position.

You can also **drag'n'drop** the selection to move it. Press ctrl before drag'n'dropping to copy the selection.

The loop

In the example, from position 4 to 8 is a loop marker:



This indicates that after the position 8 is finished, the song will loop to position 4. Note that position 9 and A are slightly dimmed, because they will never be played, unless you position the cursor explicitly on them. You can use this "out-of-boundaries" section to hide test patterns you may want to use or not later. Don't worry, unused data are optimized away on export, so don't hesitate to add seemingly useless positions.

You can edit the loop by clicking on where you want to set the start/end, or with the following shortcuts:

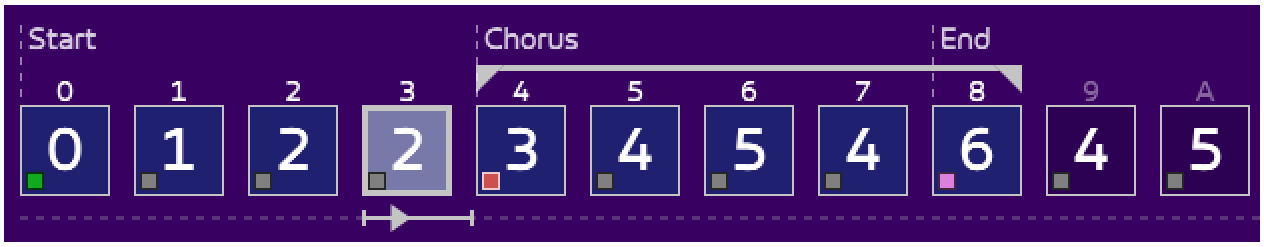
- ctrl + i to set the start where the cursor is.
- ctrl + o to set the end where the cursor is.
- ctrl + p to set the start/end from the selection of the cursor.

> Songs always loop. If you want to make a song end on a silence, create an empty pattern at the end and loop on it.

Markers

Position markers

At the very top, you can see “start”, “chorus”, “end”:



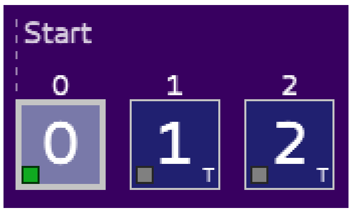
These are **position markers**, which you can use to highlight the structure of your songs for example. Edit the position to add or remove one.

Color markers

At the bottom-left on each box is a small colored rectangle. This is a **color marker** for the **pattern**, as opposed to the position markers described above. That is, if you change the color of pattern 4, you will see the change in positions 5, 7 and 9, as all refer the pattern 4. You can also click on it to edit the position (see below).

Transposition markers

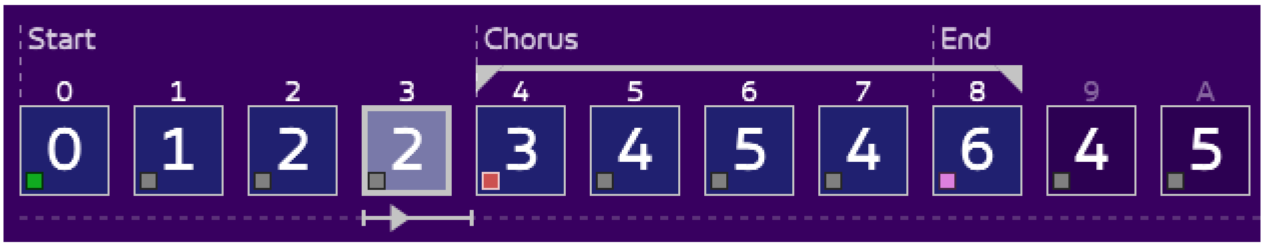
In case you transposed some tracks via the pattern viewer, a small T will appear at the bottom-right of the position box:



It is only a marker to show you that even though you may use the pattern several times, the ones marked with a T will sound differently. The transpositions are linked to the position, not the pattern, and can only be defined in the pattern viewer.

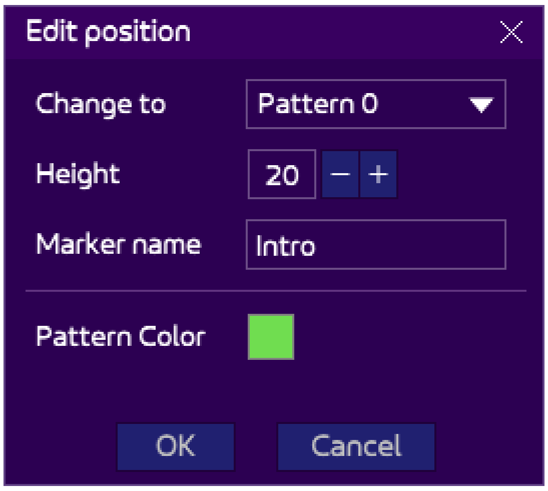
The timeline

At the bottom is a horizontal timeline that shows what is being played via a small triangle cursor. Full lines indicates what can be potentially played: past positions but also the next to come. In the following example, only the position 2 is played, because the user played its related pattern only:



Editing the position

You can edit the position by clicking on the color marker at the bottom-left of the position, or via space on a uniquely selected position. A dialog opens:


Edit position

Change to: Pattern 0

Height: 20

Marker name: Intro

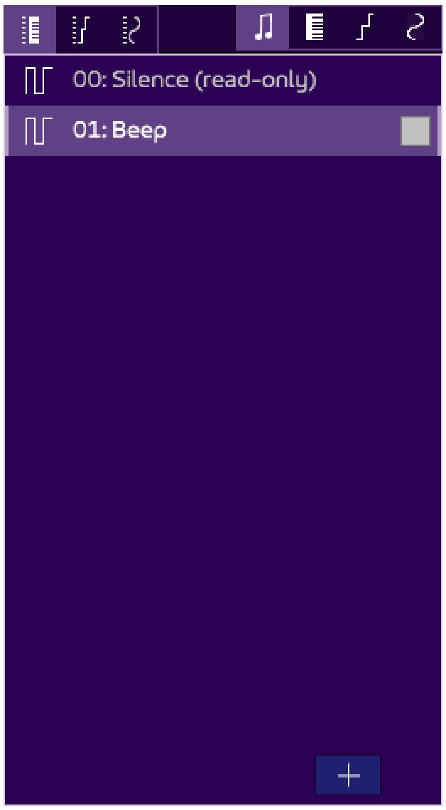
Pattern Color: 🟩

All the fields are pretty explanatory.

Notice however how the pattern color is separated from the rest: this is because it concerns the pattern, not the position, contrary to the fields above. When changing the pattern color, several positions may be impacted by this, if they use the same pattern.

The instrument and expression lists

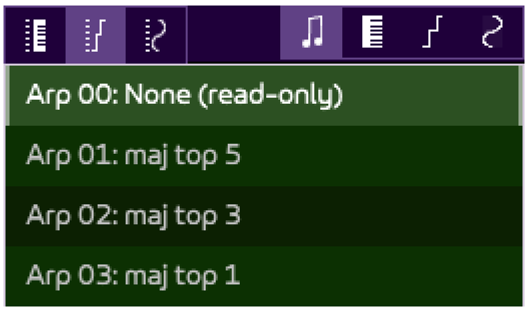
On the middle-left part of the screen are the lists of all the instruments and expressions (representing the arpeggios and pitches):


00: Silence (read-only)
01: Beep

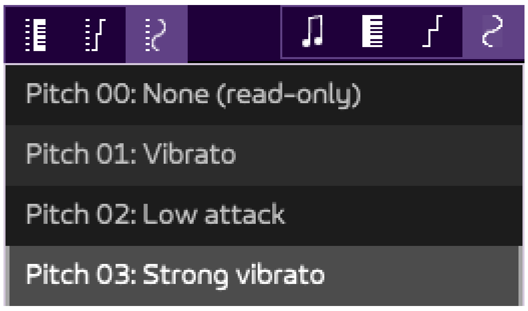
The list icons

The top-left icons concerns the list below. In the screenshot above, the first icon is selected, showing the instruments (*Beep* being the only instrument).

The second icon, when clicked, opens the arpeggio list:


Arp 00: None (read-only)
Arp 01: maj top 5
Arp 02: maj top 3
Arp 03: maj top 1

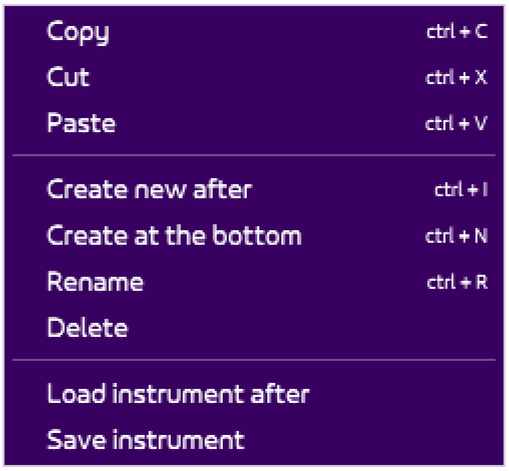
The third opens the pitch list:


Pitch 00: None (read-only)
Pitch 01: Vibrato
Pitch 02: Low attack
Pitch 03: Strong vibrato

By double-clicking on any item, or pressing enter, the related page opens.

Click the + icon at the bottom to add a new item.

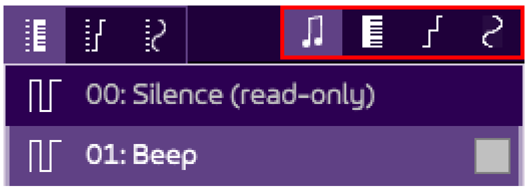
By right-clicking on an item, a contextual menu opens, with more options:


Copy ctrl + C
Cut ctrl + X
Paste ctrl + V
Create new after ctrl + I
Create at the bottom ctrl + N
Rename ctrl + R
Delete
Load instrument after
Save instrument

You can also **drag'n'drop** the items to change their order, the song will be automatically remapped with the new one. You can also multiple-select with shift or ctrl.

The right icons

At the top-right are several icons, which open panels not here, but on the larger area on the screen (where the Pattern Viewer is when starting AT):


00: Silence (read-only)
01: Beep

The icons open respectively;

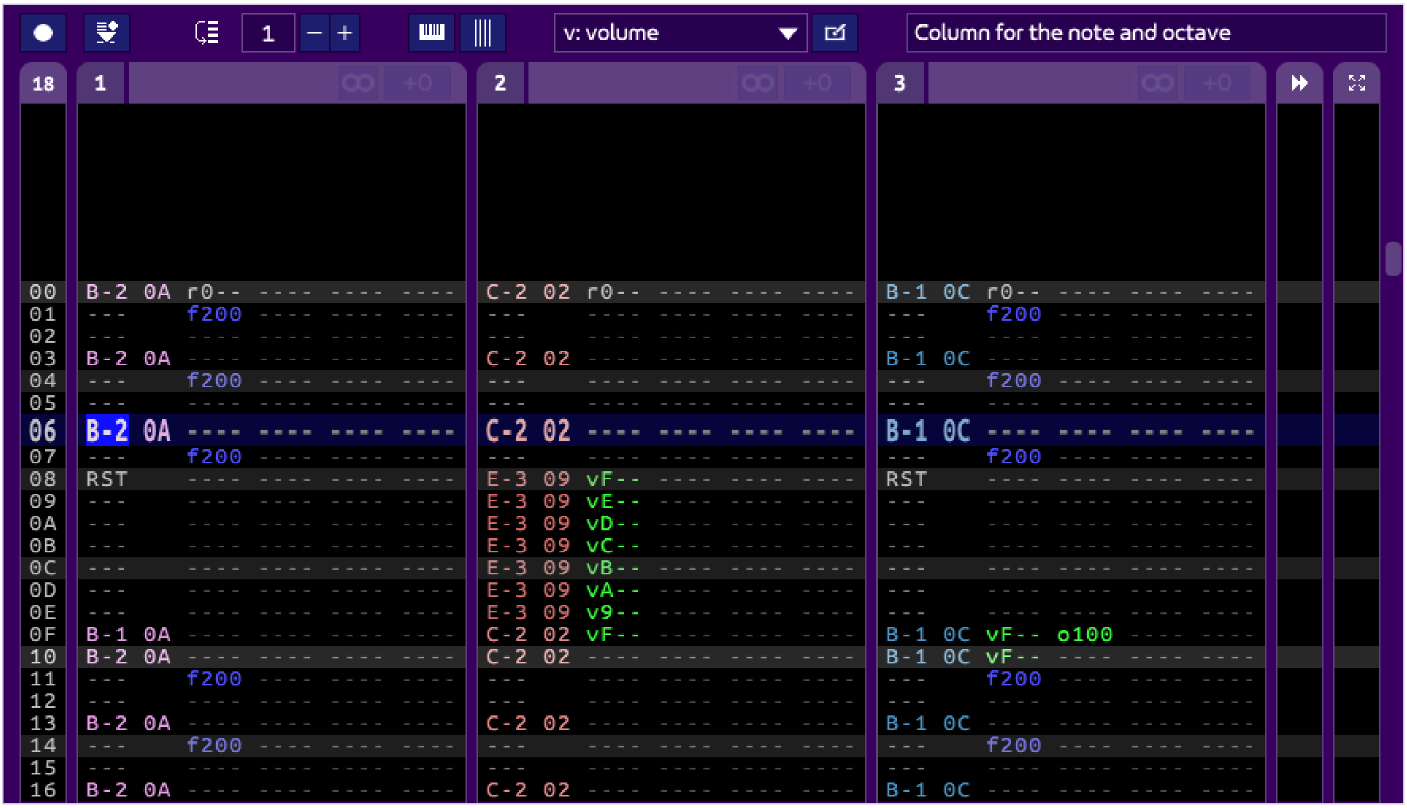
- The pattern viewer (or use F2).
- The instrument editor (or use F3).

- The arpeggio editor (or use F4).
- The pitch editor (or use F5).

Note that opening these last panels automatically opens their aforementioned list panel too. Handy!

The pattern viewer

This is the part where you can put notes in your songs. Music! The Pattern Viewer (**PV**) looks like this:



```
1   v: volume   Column for the note and octave

18  1                            2                            3
00  B-2 0A r0-- ---- ---- ----   C-2 02 r0-- ---- ---- ----   B-1 0C r0-- ---- ---- ----
01  ---    f200 ---- ---- ----   ---    ---- ---- ---- ----   ---    f200 ---- ---- ----
02  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---    ---- ---- ---- ----
03  B-2 0A ---- ---- ---- ----   C-2 02 ---- ---- ---- ----   B-1 0C ---- ---- ---- ----
04  ---    f200 ---- ---- ----   ---    ---- ---- ---- ----   ---    f200 ---- ---- ----
05  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---    ---- ---- ---- ----
06  B-2 0A ---- ---- ---- ----   C-2 02 ---- ---- ---- ----   B-1 0C ---- ---- ---- ----
07  ---    f200 ---- ---- ----   ---    ---- ---- ---- ----   ---    f200 ---- ---- ----
08  RST    ---- ---- ---- ----   E-3 09 vF-- ---- ---- ----   RST    ---- ---- ---- ----
09  ---    ---- ---- ---- ----   E-3 09 vE-- ---- ---- ----   ---    ---- ---- ---- ----
0A  ---    ---- ---- ---- ----   E-3 09 vD-- ---- ---- ----   ---    ---- ---- ---- ----
0B  ---    ---- ---- ---- ----   E-3 09 vC-- ---- ---- ----   ---    ---- ---- ---- ----
0C  ---    ---- ---- ---- ----   E-3 09 vB-- ---- ---- ----   ---    ---- ---- ---- ----
0D  ---    ---- ---- ---- ----   E-3 09 vA-- ---- ---- ----   ---    ---- ---- ---- ----
0E  ---    ---- ---- ---- ----   E-3 09 v9-- ---- ---- ----   ---    ---- ---- ---- ----
0F  B-1 0A ---- ---- ---- ----   C-2 02 vF-- ---- ---- ----   B-1 0C vF-- o100 ---- ----
10  B-2 0A ---- ---- ---- ----   C-2 02 ---- ---- ---- ----   B-1 0C vF-- ---- ---- ----
11  ---    f200 ---- ---- ----   ---    ---- ---- ---- ----   ---    f200 ---- ---- ----
12  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---    ---- ---- ---- ----
13  B-2 0A ---- ---- ---- ----   C-2 02 ---- ---- ---- ----   B-1 0C ---- ---- ---- ----
14  ---    f200 ---- ---- ----   ---    ---- ---- ---- ----   ---    f200 ---- ---- ----
15  ---    ---- ---- ---- ----   ---    ---- ---- ---- ----   ---    ---- ---- ---- ----
16  B-2 0A ---- ---- ---- ----   C-2 02 ---- ---- ---- ----   B-1 0C ---- ---- ---- ----
```

You can see three large columns in the middle: each represents a channel of your sound chip. Each column have various strange letters and numbers stacked up vertically. This is your musical sheet!

If you never experienced trackers before, the notation will probably look daunting at first, but don't worry, it's actually pretty readable and simple.

5 columns?

You can see 5 columns in the screenshots, of various width.

The one at the left, with the "18" at the top, simply numbers the lines your *pattern* (i.e. musical score) is composed of.

The three larger following columns are the actual music tracks. They are composed of *note cells* which are described below.

The next one is a speed track. This is where speed change (i.e. tempo change) are written. From the blank column, you can guess that the tempo is steady on this part of the song.

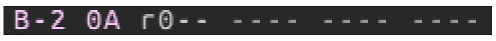
Then comes the event track . This has two uses:

- Play samples (like digidrums).
- Declare “numbers” that can be intercepted, on the real hardware, by external code. This is useful to synchronize effects to the music.

More on all these tracks below!

The note cell

Let’s take an example:



```
B-2 0A r0-- ---- ---- ----
```

The note

`B-2` is a note, using a notation mostly used in United States (but not only!), and that matches the `SI` in Solfege. Here are the matching notes:

DO	RE	MI	FA	SOL	LA	SI
C	D	E	F	G	A	B

So B-2 is a SI at the octave 2 (going from 0 to 10 , A in hex).

The instrument

The following 0A is the instrument number, going to 1 to FF (hex). The related instrument can be seen in the instrument panel on the left of the screen.

You can also see a RST somewhere in the screenshot at the top. It is actually the instrument 0, and means “reset”, cutting the previous instrument. It is useful to perform a silence. You can write one by either using the instrument 0 or use its related key (explained below).

The effects

The following `r0` is one of the many effects that can enrich your music. The list can be shown in the drop-down at the top of the panel. They are all explained thoroughly in this page.

You can stack up 4 effects per line in each track. In the example, only one effect is present. Effects may be related to the note or not, so they can be present without a note being there.

The effects are composed on the effect type ("r" for "reset"), and a value composed of one, two, or three digits. A volume effect (v) has a value of one digit, from 0 to F, whereas a pitch up (u) has a value from 0 to FFF.

The top bar

At the top of the panel can be seen several items:

Record

● Click on this to toggle the recording state. When off, you cannot modify the tracks by entering notes. When on, it goes to red, as well as the border of the panel. See below about how to edit the tracks.

Toggle record via ctrl + space

The follow mode

The next icon is the "follow mode", and indicates whether the scrolling must follow the played location, or let the user move the cursor as it wishes while the music is played. There are three modes:

This default mode make the sheet scroll when playing, but not if you are recording. This is convenient when composing: you can change notes wherever you want, and you will be able to hear the result when the playing cursor loops back to where you were.

This is the "classic" mode in all trackers (even AT2): your cursor follows the played location.

This indicates that your cursor is free to go, even when playing, and regardless of the Record state.

They can be changed via ctrl + F. More on this later.

The steps

1 – + Next are the steps. The number indicates how many lines are jumped when a note is entered. Default is 1, but you may want to increase it to, for example, 4 or 8 when you want to add a note at regular interval (such as a bass drum, snare, etc.). You can also change it via numpad / and numpad *

The other icons

Two other icons are present:

When on, the selected instrument is written alongside every note you enter. This is the default behavior, but you may want to deactivate it for two reasons:

- You have already written a sequence of notes and instruments, and only want to replace some notes here and here without bothering to change the instrument every time.
- You want to write a *legato*: a note without an instrument. When a note is played without instrument, the instrument doesn't restart, but continue with the new note.

This shrinks/enlarges the track, by hiding the values of the effects. This is useful when working on a small screen and having many tracks.

The effect selector

A drop-down and an icon follows:

v: volume

This is the effect to write if you enter a value in an effect column. You can write it yourself if you want to, but by selecting it in the drop-down, you don't have to remember its related letter (v for volume, etc.), and it also saves some time and you only need to enter the value, not the effect.

Click on the drop-down to chose any effect to use.

Note that by default, effects type are not overwritten. Check the following icon, which can be toggled.

This is selected by default. When typing an effect value, the possibly already present effect type is not overwritten. For example, if a pitch down ("d") is present, it won't be overwritten by your the selected effect type in the drop-down.

If selected, the selected effect overwrites the effect type that can be present.

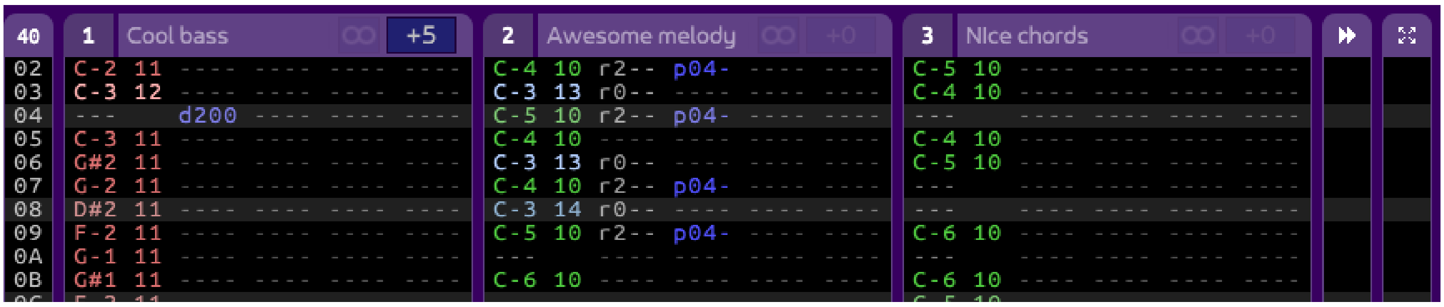
The information view

Column for the note and octave

The last item indicates what is under the cursor. If over an instrument, it will shown its name. If under an effect, it will show its use.

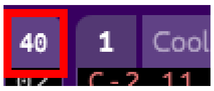
The track headers

Each column has a header, which may look like this depending on your song:



```
40   1 Cool bass              +5   2 Awesome melody         +0   3 NIce chords            +0
02   C-2 11 ---- ---- ---- ----    C-4 10 r2-- p04- ---- ----    C-5 10 ---- ---- ---- ----
03   C-3 12 ---- ---- ---- ----    C-3 13 r0-- ---- ---- ----    C-4 10 ---- ---- ---- ----
04   ---    d200 ---- ---- ----    C-5 10 r2-- p04- ---- ----    ---    ---- ---- ---- ----
05   C-3 11 ---- ---- ---- ----    C-4 10 ---- ---- ---- ----    C-4 10 ---- ---- ---- ----
06   G#2 11 ---- ---- ---- ----    C-3 13 r0-- ---- ---- ----    C-5 10 ---- ---- ---- ----
07   G-2 11 ---- ---- ---- ----    C-4 10 r2-- p04- ---- ----    ---    ---- ---- ---- ----
08   D#2 11 ---- ---- ---- ----    C-3 14 r0-- ---- ---- ----    ---    ---- ---- ---- ----
09   F-2 11 ---- ---- ---- ----    C-5 10 r2-- p04- ---- ----    C-6 10 ---- ---- ---- ----
0A   G-1 11 ---- ---- ---- ----    ---    ---- ---- ---- ----    ---    ---- ---- ---- ----
0B   G#1 11 ---- ---- ---- ----    C-6 10 ---- ---- ---- ----    C-6 10 ---- ---- ---- ----
```

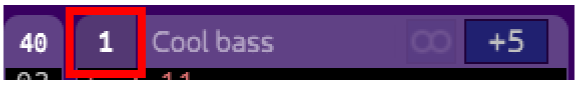
Position height



Each position has a height, that is, how many lines you can see and hear. Default is 64 (40 in hex, as seen above), with a maximum of 128 (80 in hex). By clicking on the displayed height, a pop-up will show and you will be able to modify it.

As explained in the Linker page, each position has its own height, independently to the pattern it uses. So this allows using the same pattern multiple times, but with a different height.

Toggle on/off

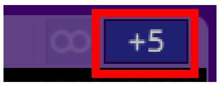


Each track can be muted, or you can also solo one (muting all the others). You can mute/unmute it by left-clicking on the track number, or solo/unsolo it by right-clicking. The same can be done on the EQ on the top-left of the screen.

You can also use the keyboard shortcuts:

- ctrl + numpad 0 to mute/unmute the track where the cursor is.
- ctrl + numpad 1 to solo/unsolo the track where the cursor is.

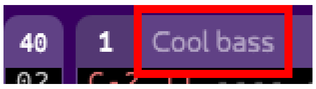
Transposition



A transposition allows to add/subtract semi-tones to a track, making it sound “higher” or “lower” without having you transpose the notes manually. This is also memory-saving as AT can encode the tracks with the same data.

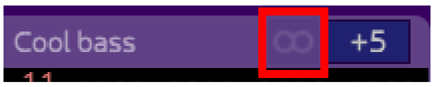
As explained in the Linker page, each track of each position has its own transposition. So you can re-use the same pattern multiple times and yet have their tracks sound differently thanks to the transposition. Let's emphasize on this again: even though presented in the **pattern** viewer, transpositions are related to the **current** position.

Track names



Each track can be named independently. This can be for fun, or useful if you need to structure them in your mind. A third possibility is **linking**, as seen just below.

Linking



Click on this icon to "link" or "unlink". This is an **advanced** feature to re-use tracks throughout the songs without having to copy/paste them. This was the *de facto* way of working in STarKos, AT1 and AT2, but could become soon overwhelming, though it could help optimize the size of your song. AT3 does that automatically for you, which works well enough for more cases. You can learn more about this feature here.

Playing and editing

Click anywhere in the music sheet. Notice how your cursor goes there. You can use the cursor keys to move it in all directions, or page updownendstart to move quickly.

Playing note, as in most trackers, is done is done via your computer keyboard. Select first any instrument on the list at the left of the screen, and type on keys such as Q, W, E, R, for the upper octave, or Z, X, C, V for the lower octave (on a QWERTY keyboard). The "upper line" of your computer keyboard represents the higher octave, the "lower line" the lower.

> If using the computer keyboard bothers you, you can use any external MIDI controller instead! Just set it up and use it instead!

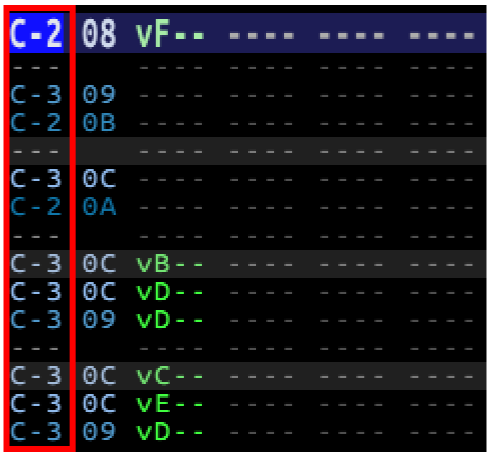
> You can change to the previous/next instrument using alt + up or alt + down.

Note that, for now, you can hear the sound but this does not add any note. Click the Record icon ●. It goes to red, as well as the border of the PV. This means that entering notes will not only play them, but also write them.

It is important to know that entering a note can only be done in a Note column. The other columns have a different semantics, and the QWERTY/AZERTY keys will behave differently). Let's now explain all these columns.

The note column

Click on a track, in the Note column (this is the left-most column), as highlighted in the red border in the following image:



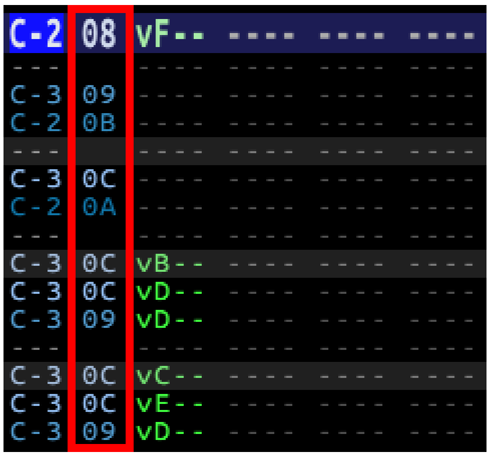
```
C-2 08 vF-- ---- ---- ----
---    ---- ---- ---- ----
C-3 09 ---- ---- ---- ----
C-2 0B ---- ---- ---- ----
---    ---- ---- ---- ----
C-3 0C ---- ---- ---- ----
C-2 0A ---- ---- ---- ----
---    ---- ---- ---- ----
C-3 0C vB-- ---- ---- ----
C-3 0C vD-- ---- ---- ----
C-3 09 vD-- ---- ---- ----
---    ---- ---- ---- ----
C-3 0C vC-- ---- ---- ----
C-3 0C vE-- ---- ---- ----
C-3 09 vD-- ---- ---- ----
```

Enter a note with the keyboard (computer or MIDI). This will and play it and write it where the cursor is! Note how the instrument number you selected (`01` for example) is written on the right of the note. As said above, you can disable the instrument writing with the Instrument icon.

Writing also makes the sheet scroll down of one line, which is handy to enter notes one after the other. How many lines are jumped is set by the steps, as explained above.

The instrument column

The instrument is an hexadecimal number from 01 to FF. 00 is the "stop sound", also known as RST (Reset), which you can write with shift + R.

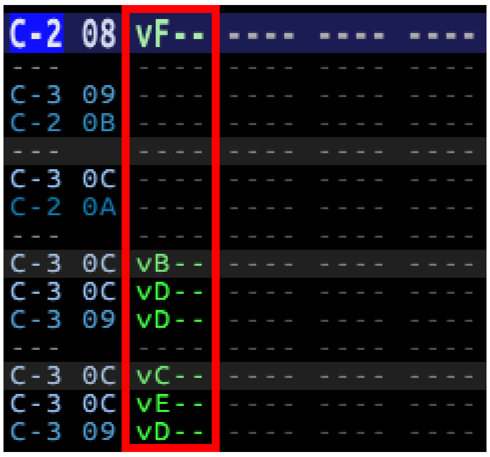


```
C-2 08 vF-- ---- ---- ----
---    ---- ---- ---- ----
C-3 09 ---- ---- ---- ----
C-2 0B ---- ---- ---- ----
---    ---- ---- ---- ----
C-3 0C ---- ---- ---- ----
C-2 0A ---- ---- ---- ----
---    ---- ---- ---- ----
C-3 0C vB-- ---- ---- ----
C-3 0C vD-- ---- ---- ----
C-3 09 vD-- ---- ---- ----
---    ---- ---- ---- ----
C-3 0C vC-- ---- ---- ----
C-3 0C vE-- ---- ---- ----
C-3 09 vD-- ---- ---- ----
```

You can also find note without instruments. This is called *legato*, and means that the note is changed but the instrument not restart. You have two ways of creating this:

- either go to the instrument column and press delete to remove the instrument
- or disable the aforementioned Instrument write and write your note, without an instrument.

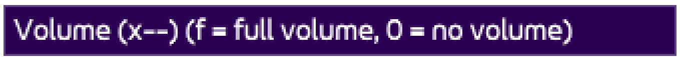
The effect columns



```
C-2 08 vF-- ---- ---- ----
---    ---- ---- ---- ----
C-3 09 ---- ---- ---- ----
C-2 0B ---- ---- ---- ----
---    ---- ---- ---- ----
C-3 0C ---- ---- ---- ----
C-2 0A ---- ---- ---- ----
---    ---- ---- ---- ----
C-3 0C vB-- ---- ---- ----
C-3 0C vD-- ---- ---- ----
C-3 09 vD-- ---- ---- ----
---    ---- ---- ---- ----
C-3 0C vC-- ---- ---- ----
C-3 0C vE-- ---- ---- ----
C-3 09 vD-- ---- ---- ----
```

There are four columns of effects, which is more than enough to create rich and expressive music. Most of the times, only 1 or 2 are used. Each column is made of one letter (the effect type), and 0 to 3 digits, forming the effect value. How many digits depends on the effect. Remember that all these digits are written in hexadecimal, so a 3-digit value would go from 000 to FFF.

For example, go to the left column of the effect, put the Record mode to on, and type "v". A first default digit (0) will also show. By going back to the effect type, the information column shows:



```
Volume (x--) (f = full volume, 0 = no volume)
```

It means that only one digit is used, and its value going from 0 (no volume) to F (full volume). Go to the first digit and type F. You should hear the note with a full volume! You can change the value to A, 8, 4, etc. and notice how the note volume changes.

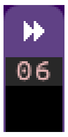
All the effects are explained here.

Knowing which key triggers what effect is somewhat bothersome. Which is why the effect drop-down is useful. Also, the effect keys and colors can both be customized in their respective setup!

The effects have an order, which is only limited by logic. For example, two Volume effects is a mistake, as the first would be overridden by the second. When something illogical like this happens, the effect is displayed in red, and an explanation is displayed in the information view.

The speed track

At the right of the panel are two smaller columns. The first is is the Speed Track.



It holds number from 01 to FF, which you can edit like any other column. The purpose of this track is to define the speed of the song, 01 being *very fast*, FF *very slow* (06 being default and mid-tempo). The speed changes whenever the playing cursor finds a new speed value. It is thus possible to change the speed at any time.

In fact, the speed is not a “real” speed. It is actually a duration, how many *frames* the player *spends* on each line of the tracks.

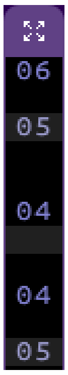
So a speed of 6 (with a player running at 50 Hz) means that each line will last 0.12s (1 / 50 Hz * 6).

If your song does not change its speed, you can ignore this speed track, yet define the start speed in the parameter of your subsong (Edit > Subsong properties > (select your subsong) > Initial speed).

More explanations about speed and BPM on this page, including shuffle and technicalities!

The event track

The full-right track is the mysterious “event” track.



In most songs, this will be empty. This track has two purposes, most of the time exclusive:

- Send events to the player
- Play samples! This madness is explained below.

Events

Imagine your music is for a demo, and you'd like to synchronize effects with what is going on in your song. Timing all this is tiresome and could require many trials-and-errors. Events come to the rescue!

By adding simple numbers, from 01 to FF, you can have the player intercept them and react accordingly. You could for example define with the programmer that 01 means "start of the second effect", or "red flash on the screen", 02 being "blue flash" etc. It's all up to you.

To learn on how to intercept events, please check this page.

> If you also use samples, it is **strongly** advised to use the events from the top numbers (FF) and going downwards to avoid clashing with the sample numbering.

Samples

Yes, the event track can also trigger samples. Simply write the instrument number to play, and here you go. But why use this track to play samples instead of the conventional "music" tracks?

- This is mostly useful for digidrums. You don't have to select a specific note, only the instrument.
- Samples played in event tracks have priorities over the conventional PSG sounds, just like your sample player would probably do. You basically don't want your samples being replaced by normal sounds. The samples are basically heard "over" the other sounds.

One drawback is that such samples can only be played on one channel, always the same per subsong. You can define it in subsong properties: Edit > Subsong properties > (select your subsong) > Digichannel.

Playing

While not recording, the aforementioned keys on your computer keyboard (QWERTY for a Qwerty keyboard) will make sound, on the channel where the cursor is, and the selected instrument on the list on the left of the screen.

Use numeric pad + and numeric pad - to change the octave, displayed at the top of the screen.

When the pattern viewer is focused, playing with the keyboard will use the cursor location to determine on which channel to play, and its current effects (volume, arpeggios and so on). If any other part of the screen is focused, the Test Area is used. Learn more about this in its related page.

The general keys to play the pattern applies:

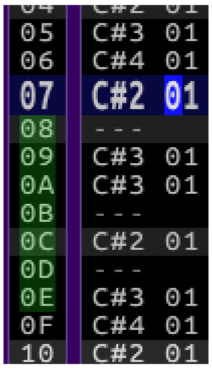
- F9 to play the song from the start.
- F10 to play the song from where you are.
- F11 to play the pattern from the start.
- F12 to play the pattern from where you are or from the block start (see below).
- space to play the pattern from the *block* start (see below) if it exists, or from the start of the pattern if the block does not exist.
- esc to stop.

But some keys are specific to the PV:

- ctrl + W to play the pattern from the start.
- shift + space to set the block start from where you are, and play.
- esc to both stop playing and clear the block if not playing.

Block loop

The *block* is a new and very handy feature. It is simply a part of the pattern to be played in loop. You can create it via the keys above, or draw it by yourself by drag'n'dropping with the mouse on the line column on the left:



```
05  C#3 01
06  C#4 01
07  C#2 01
08  ---
09  C#3 01
0A  C#3 01
0B  ---
0C  C#2 01
0D  ---
0E  C#3 01
0F  C#4 01
10  C#2 01
```

The next time you will press space, the lines from 08 to 0E will loop! To override it, you can either press esc to clear it, or shift + space to set the block start from where you are, and play.

When a block is one line high (less that four will only validate the first one actually), the pattern will play from the block to the end of the pattern, and loop at the block start. If the block is four lines or more, it will loop over its height.

Looping over a section is nice, but what comes handy is using another Follow mode, or the Record mode to on.

Editing

Playing note, or trying to modify them will not do anything unless Record mode is on. So with the PV focused, click on the record button or press ctrl + space. The border turns red. Using your computer keyboard will now write note.

What the keyboard will do depends on where your cursor is. Move your cursor either by left-clicking with the mouse on a digit or empty space, or with the keyboard cursor (left or right to go from a column to another, up or down to go one line above or below.

Some handy shortcuts available on any columns:

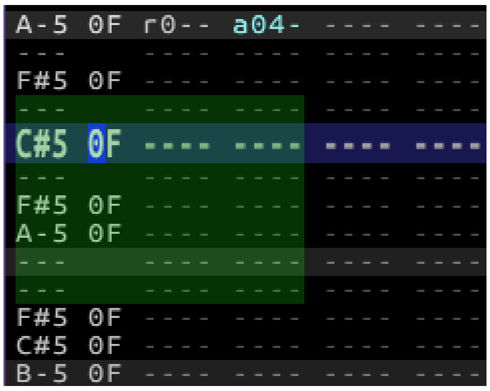
- del to clear anything under the cursor.
- insert to insert a line.
- suppr to delete a line.
- enter to hear the notes on the full line and go to the next one.
- And the usual cut/copy/paste/undo/redo.

The note, instrument and effects columns have all been described above.

Selection

By left-clicking and dragging, you can create a selection, which can then be copied (ctrl + C), paste (ctrl + V), cut/cleared (ctrl + X), but also transposed (ctrl + up, ctrl + down).

Note that your cursor is considered a selection, so all the shortcuts here works even when nothing is apparently selected. Moving your cursor also dismisses the selection.



```
A-5 0F r0-- a04- ---- ----
---    ---- ---- ---- ----
F#5 0F ---- ---- ---- ----
---    ---- ---- ---- ----
C#5 0F ---- ---- ---- ----
---    ---- ---- ---- ----
F#5 0F ---- ---- ---- ----
A-5 0F ---- ---- ---- ----
---    ---- ---- ---- ----
---    ---- ---- ---- ----
F#5 0F ---- ---- ---- ----
C#5 0F ---- ---- ---- ----
B-5 0F ---- ---- ---- ----
```

Transposing allows to increase/decrease the note value by semi-tones. Adding shift to the combination increases the transposition rate (an octave).

One nice feature of Arkos Tracker is that is it possible to *transpose* not only notes, but instruments and effects as well, which is handy to increase effects values because you deemed them too slow for example. This is also handy to change a value without having to type it manually.

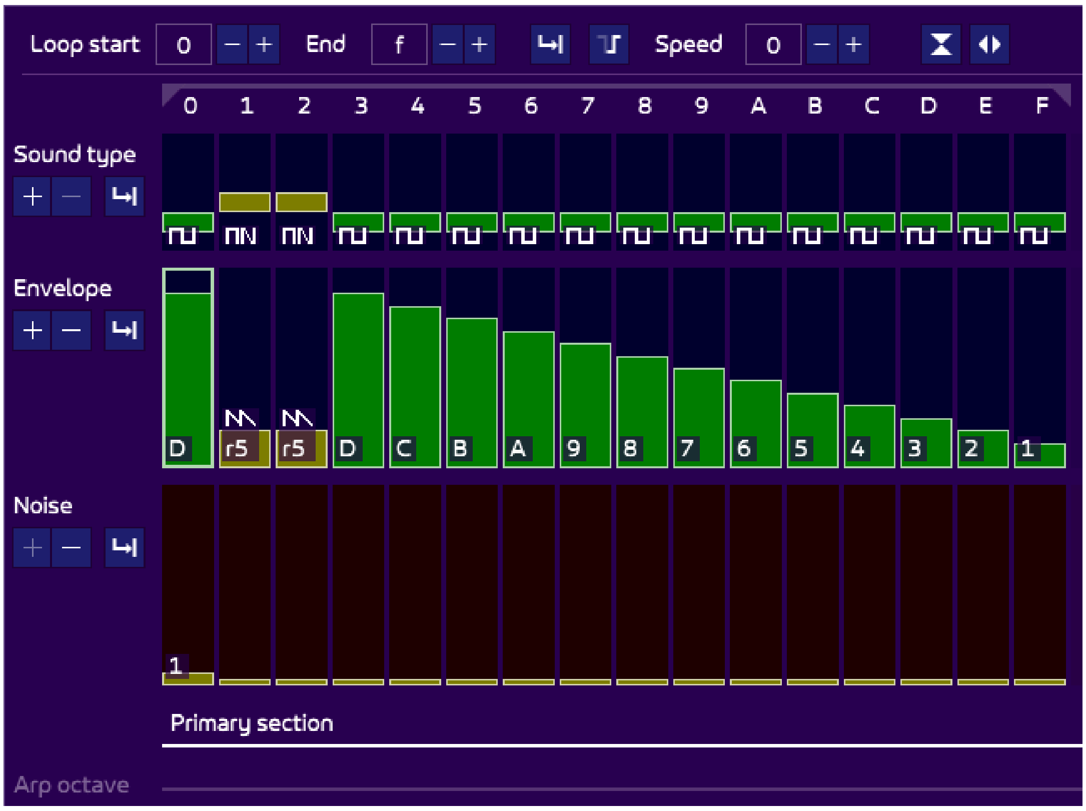
Such selection must not include notes, as notes have priorities in the selection. Indeed, if you select a whole track, it would be illogical that the transposition affects the notes, but also all the effects values.

The instrument editor

> Before reading further, it is strongly advised to have basic knowledge of what the AY/YM can do and how its sounds are produced. I advise you to read this wonderful tutorial, yay.

As its name implies, the Instrument Editor (**IE**) allows to create and modify the instruments of your song. In the olden days of STarKos, and Arkos Tracker 1 and 2, creating your instruments was done via a complex Excel sheet-like full of numbers, which might have scared many users.

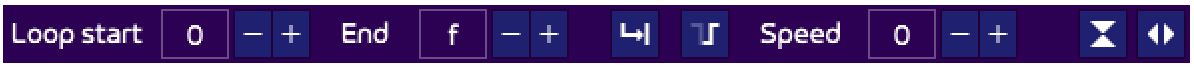
The good news is that the IE has been completely overhauled in AT3, adding more comfort, ease of use, and even more power here and there. Let's have a look at it:



Everything is meant to be editable via the **mouse**. However, experienced users will find it faster to use the **keyboard** and edit everything manually, which is also possible. More on this below.

Columns and loop

The sound is composed of **columns**, noted from 0 to F here (but possibly up to FF), and read from left to right, each representing a frame. Logically, the more columns, the longer the sound. If your player runs at 50 Hz (as it most of the times is, and is default), meaning 50 frames per seconds, then one column will be played every 50 Hz. As an example, the sound above has 16 columns (0 to F in hex), it will thus last about a third of a second.



At the top, you can see an **end** slider. This indicates the last column to be played before the sound stops (or loops, as we're going to see). So it is possible to have many columns, but an end much shorter, which is handy for experimentation.

The **loop start** slider indicates where to loop when the end is met, provided that you allowed the sound to loop. This is done via the **loop** icon on the top middle. Clicking on it switches from not looping to looping .

A looping sound will play **endlessly**.

The next icon deals with a feature called **retrig** and is specific to hardware sounds. More on this later.

Then is the **speed**. I told you earlier that one column is played every frame. Very often, you will find that it is too fast: if you want a slow progressing sound, this would require you to duplicate columns to slow it down. This is both unpractical and memory consuming. The speed counter allows you to slow down the sound: the higher the speed, the slower the sound. A speed of 0 is the fastest: one column per frame. A speed of 1 will play one column for two frames, 2 for 3 frames, 3 for 4, etc.

The second last icon is the **shrink** icon . It is purely visual, and allows to hide rows when their data consists only of default values. It is explained in detail below.

The last icon is the **zoom** icon and is also purely visual: it toggles the horizontal zoom of the display, which can be handy if your sound has any columns.

Playing

What would be an Instrument Editor if you couldn't play your sounds, eh? Just like in the Pattern Viewer, or anywhere in the software, simply use your MIDI keyboard, or your computer keyboard

(QWERTY or AZERTY, with numeric pad +numeric pad - to change the octave). More on this Test Area page.

Editing the values

Before showing the meaning of all the rows, it is important that you know how to edit the values, test things, and have fun! The in-depth explanations of the values, in the following section, can wait for now!

There are basically two ways to edit the values. With the mouse, or via the keyboard.

Editing via the mouse

This is the simplest way of edition, since you move your mouse on a row and “draw” the desired value.

- mouse wheel up mouse wheel down to increase/decrease the value. Add shift or/and ctrl to increase the value even more.
- ctrl + left-click to draw the value where your mouse is.
- left-click to set the cursor position.
- right-click to open a context menu with more options. **Note:** these options work from where the cursor is (not the mouse cursor).
- double-left-click to set the value. See below.

Also don’t forget these to navigate:

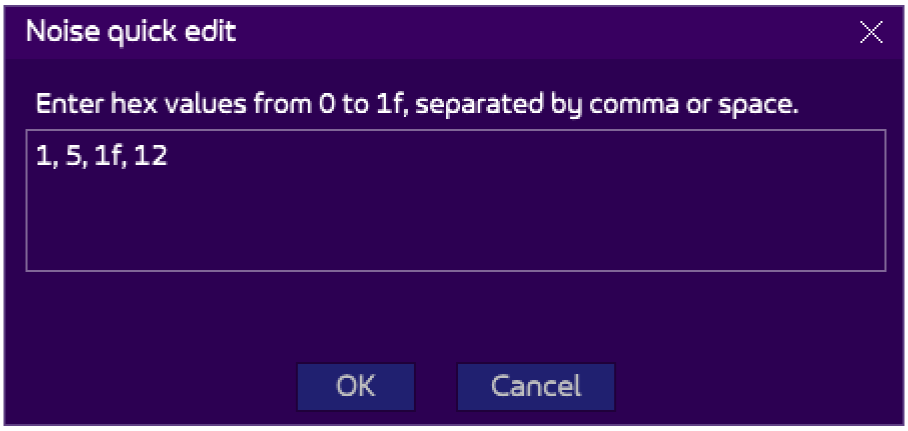
- mouse wheel to scroll vertically.
- alt + mouse wheel to scroll horizontally.
- alt + up alt + down to change to the previous/next instrument.

Editing via the keyboard

A more precise and maybe faster way is to use the keyboard, which you can see at the top-left of the instrument.

- cursor to move your cursor around.
- page up page down to move the cursor vertically.
- start end to move the cursor to the start/end of the instrument horizontally.
- shift or/and ctrl + cursor up/down to increase/decrease the value.
- enter to edit the value(s) manually.

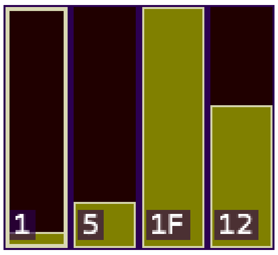
The latter shortcut opens a dialog. By following the instructions, you can edit one, or many values. Example with the noise:


Noise quick edit

Enter hex values from 0 to 1f, separated by comma or space.

1, 5, 1f, 12

Resulting into this:

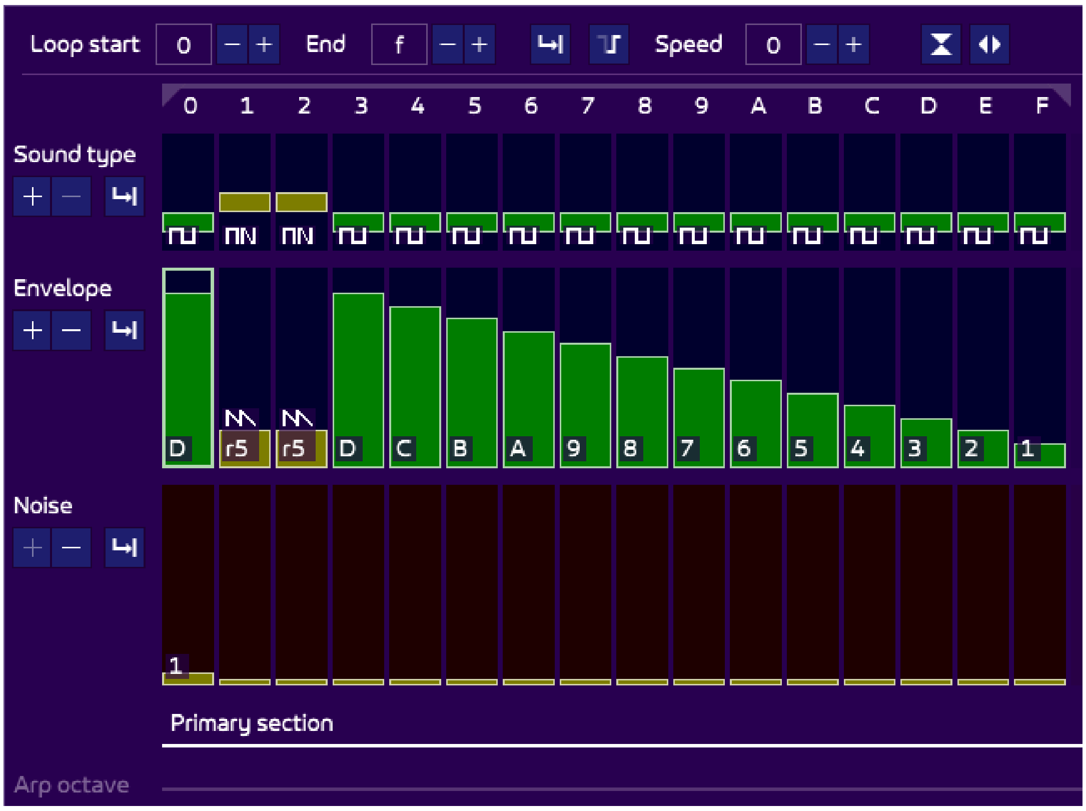


You can use space as a separator (`1 5 1f 12`), or comma (`1,5,1f,12`) or both. Any wrong values will be explained via a message.

More complex values, for sound types and envelopes, will have more extensive instructions, but it is basically the same. This way of editing allows you to quickly add values if you have a certain vision of the sound you want. You can also type random values and see what is happening!

Row meanings

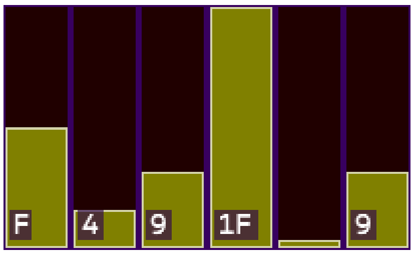
Now that's you've seen the columns, how to move between them and edit the values, you must understand what all the rows are about. This is the most interesting part, but also where more explanations are needed.



Cells

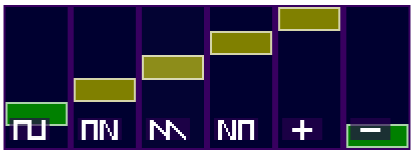
Rows are composed of cells. They all follow the same design.

A vertical bar represents a value (the taller, the higher the value), and at the bottom is an image or a text representing the value. A simple example is the noise:

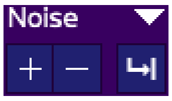


Very often, values of 0 are not displayed to avoid cluttering the interface with useless data.

But some cells may be using an image, such as the sound type:



On the left of every row is the row header:



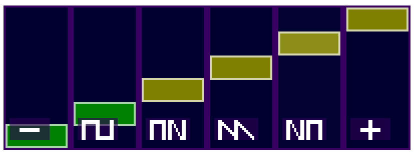
You can use + or – to visually increase or decrease the height of the row (and thus, the bars). This has an implication: tall bar will take more space, but also may show more values, which actually may not be useful, because by default the bars shown the most useful values, discarding the “useless” ones. More on this later.

There is also a specific loop icon called **auto-spread loop**, which is a handy and new feature to automatically copy/paste values without you having to do it manually. This is explained below.

A last option, available in most rows, is the ability to hide the row via a small downward pointing arrow. Some rows are indeed not often used, and you may want to hide them for more clarity. More on this below.

Sound type

This row is at the top because it is the most important and will define how some rows below will behave. Once again, please check the tutorial to understand better how the PSG works. You can set the type to 6 different values, from minimum to maximum. From left to right:

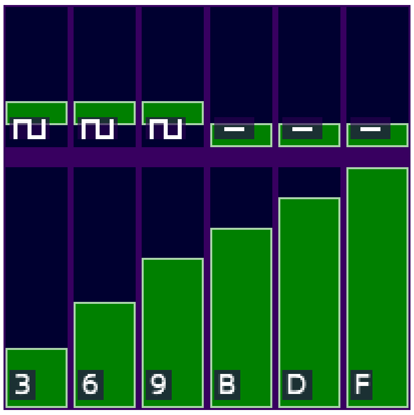


- 0: **no software/no hardware**. This means that there will be no sound, either “software” (the sound channel is closed: no square wave is generated), or “hardware” (the hardware envelope is deactivated). The only thing you could hear would be the noise generator.
- 1: **software only**. The sound channel is open: the square wave is generated. This is the classical PSG sound from the 80s.
- 2: **software to hardware**. This is the typical “hardware” sound everyone knows. The square wave is generated, coupled with the hardware envelope. The hardware wave period is calculated from the software period, which in turn comes directly from the note of the musical sheet. So the software “leads” (more on this in the next row explanation).
- 3: **hardware only**. No square wave is generated, only the hardware envelope. Such sounds are not often used. They are often softer than the “software to hardware” sound, perfectly constant.
- 4: **hardware to software**. Like “software to hardware”, except that the hardware period “leads” the calculation of the software. As a result, it sounds as full as “software to hardware”, but it is more stable, because there is no rounding in the calculation between the periods (a multiplication is used). However, the hardware period being more accurate, the software period lose some accuracy and the resulting sound may be out of key as you go into the higher notes. That is why such sound is mostly used for bass. An option, explained later, exists to de-sync the waves to create a “mwwwaaahhh” sound as you desire.

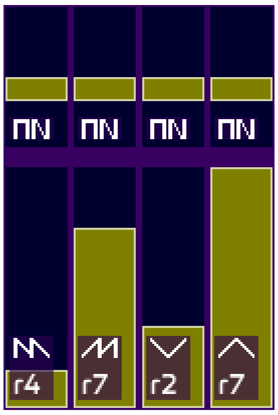
- 5: **software and hardware**. The final option is also the less used, though it allows special and original effects. It allows the software period and hardware period to be both fed with the input period (the note from the music sheet), but there is no calculation between the two, they are not linked. You can add arpeggios to each if wanted. Icing on the cake, you can force a period on one and not the other. Typically, this allows some Ben Daglish effects where the software wave is used, but the hardware period is forced to a very high frequency (i.e. low period). Just listen to the H.A.T.E. or Skate Crazy solos for example!

Envelope

This is the second-most important row, and diverts directly from the sound type. In the two "software" sound types (software only/no software no hardware), the bar represents a volume from 0 (mute) to F (full):



As soon as a hardware envelope is involved, it becomes a bit more complex:

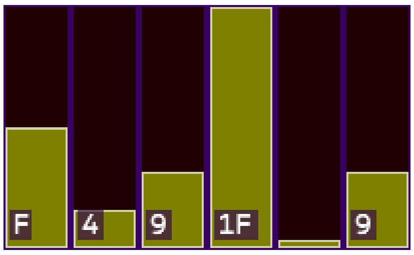


- A hardware envelope icon is shown. It shows the evolution of the volume, generated by the hardware envelope generator, producing from the left to right columns: a sawtooth (F to 0, looping), an inverted sawtooth (0 to F, looping), a triangle (F to 0 to F, looping), and an inverted triangle (0 to F to 0, looping). All the others envelope, less used, are hidden by default (because not very useful besides special effects), but you can see them by maximizing the height of the row to see them.

- A ratio (from 0 to 7) may be shown (r0, r1, etc.). It is only present when both waves are coupled (such as “software to hardware” or “hardware to software”). The ratio is used to calculate how the **secondary** period (hardware for “software to hardware”, software for “hardware to software”) is calculated from the first (the **primary**). Basically, after the primary period is calculated, it is divided by 2 to the power of “ratio” for “software to hardware”, or multiplied by 2 to the power of “ratio” for “hardware to software”. It may sound complicated, but it is not: just consider that one period leads, and the other is calculated with the help of a ratio, which drastically changes how it sounds. Just experiment and have fun! The higher the ratio, the bigger the operand, so the result may sound either unmusical, or spacey. A ratio from 3 to 5 is mostly used, with 4 being the champion (albeit probably over-used) .

Noise

Noise is mostly used for drums and special effects (explosions etc.). Its value goes from 1 (high-pitched noise) to 31 (low-pitched noise), 0 meaning “no noise”.



Noise can be used regardless of the sound type. It can be used with hardware sounds (sounds mostly bad, though), software sounds and even “no sound”. For the last two, just make sure there a volume of at least one.

Primary/secondary sections

These sections might intrigue you, especially if you’ve used STarkos or AT1/2 before. Their purpose will become clearer when reminding you how the sound types are named:

- No software/no hardware
- Software only
- Software to hardware
- Hardware only
- Hardware to software
- Software and hardware

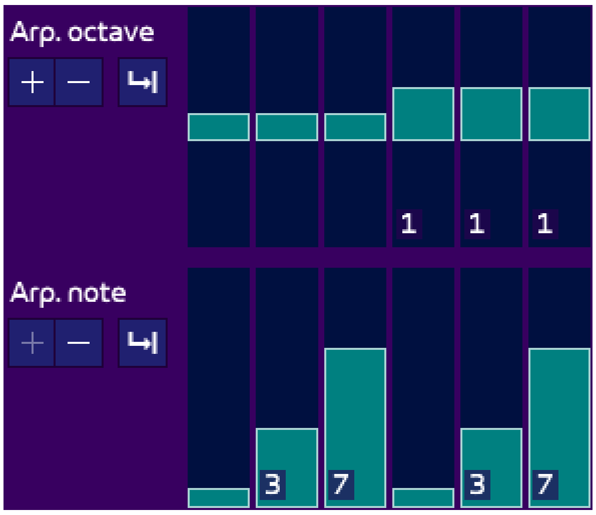
Most of them have only one or no “side”: no software/no hardware, software only, hardware only. This means that their data is only in the primary section (i.e. the first one).

The others have two sides, with a possible direction: software **to** hardware, hardware **to** software, software **and** hardware. The primary section is the left-side of the sound type name, the secondary section is the right-side. So for example, the primary section of "hardware to software" is for the hardware-side, the secondary section is for the software-side.

Both sections contain the same rows: arpeggio octave, arpeggio note, pitch, and period, described just below.

Arpeggio octave and note

By adding an arpeggio to the base note, you simply add semi-tones to it. In order to visualize this better, AT3 splits the arpeggios into octave and note. You can thus concentrate on the chords in the "Arp. note" section, and brighten it up with the "Arp. octave". Example:



Here you can see the arpeggio notes "0 3 7 0 3 7" (0 being hidden because not important). This is a double-minor chord (0 4 7 would be a major chord). But the second chord is also one octave above.

Adding octave is especially useful along with hardware sounds. Some ratios sound better with higher or lower octaves, so don't hesitate to play with different combinations.

You may also wonder why you should use arpeggios in the instrument whereas you can also add them via arpeggio effect in the pattern viewer. This is an excellent question! And the answer is two-fold:

- As said just above, you might want to tweak the octave for specific hardware columns, which would be tedious to do in the pattern.
- Arpeggios in the instrument and in the pattern are **added**. Which is awesome because you can create very organic sounds by mixing the two. Advice: use a change of octave in the instrument, and use a "melodic" arpeggio (0 3 7, etc.) in the pattern. Since the instruments and the arpeggio will probably have different lengths, the sounds will be richer and progress a lot more than if it was only an arpeggio in the instrument or in the pattern.

Pitch

The pitch is a simple addition to the period of the sound. You can create vibratos with this (though you should rather use the pitch effect in the pattern). Pitch is also very useful in drums (a pitch-down emulates a bass-drum or a tom for example).

A special effect can be done in the *secondary* section, if you are using a “software to hardware” or “hardware to software” sound: you can add a desync by yourself using the pitch. This is especially useful in the “hardware to software” sound, where both envelopes are originally perfectly synchronized, thus created a very long “mwaaaaaahhh” sound, or any special effect you can come up with.

Period

This is probably the less-used row of all, yet you can come up with interesting effects. It basically forces the period, thus ignoring the one from the musical sheet. A value of 0 means “automatic”, the default case in which AT will calculate the period from the musical sheet, and the aforementioned arpeggio and pitch.

However, setting the period can be used for these purposes:

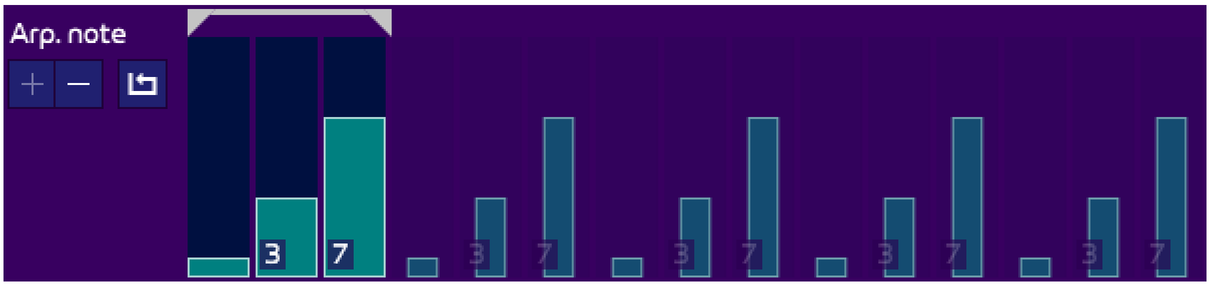
- For a drum, you could hand-write the periods you want. One advantage is that you can then transpose any track using your instrument, it will always sound the same.
- You can force the period for a drum and have the rest of the sound use the automatic period: the beginning of the sound (for example) will always sound the same, the rest will follow the notes from the pattern. Mad Max on Atari ST used this technique a lot to create bass preceded by an attack of a drum.
- If using the “software and hardware” sound, force the period of the secondary period to a low value (from 2 to 10) and enjoy a “Ben Daglish” sound!

Note that imposing the period renders the arpeggio and pitch rows of the section useless: they will be greyed out. Indeed, if you force the period, nothing can be calculated from it.

Also, be mindful that forcing the period will sound right only on your target hardware: you will have to adapt it for it to sound the same on a hardware with a PSG of a different frequency.

Auto-spread loops

This is a new feature in AT3. Remember how you sometimes want to spread a few values of a row (noise, volume, arpeggio, etc.) up to the end of sound? There was a “spread” option in AT2, but every time you made a change, you had to "spread" again. AT3 does this automatically. Imagine you have a 0-3-7 arpeggio you want to see during the whole sound. Simply enter it once, click on the "auto spread icon" on the left of the header of the row. A loop handle appears at the top:



You can set the auto-spread loop by clicking on the start/end of the loop handle, or use the cursor and the following keys:

- ctrl + shift + p to toggle the loop.
- ctrl + shift + i ctrl + shift + o to set the start/end where the cursor is.

In this example, the loop is set to the first three columns, and the arpeggio is copied/pasted till the end of the sound automagically! Modifying the looped sequence automatically changes the spread values. This is non-destructive: you can click again on the loop icon to disable it, and the previous values will reappear. The generated values, slightly greyed-out and shown with thinner bars, cannot be modified.

Note that this auto-spread loop is **only** a way to quickly and automatically copy/paste values. It does **not** create an inner loop that would be different from the loop of the instrument.

Also, you cannot spread up to a specific column. The values are spread up to the end of the sound.

Retrig

Retrig is an slightly advanced concept and **only** concerns hardware sounds. Basically, when you ask for a specific hardware envelope (8, sawtooth for example), then ask for it again later, AT will ask the PSG to continue where it was and will not trigger the envelope from the start. This may be a bit bothersome if you want to force an attack when a new note is sequenced in a pattern: if the same note is played, it won't make the difference with the previous one. Since you don't control the volume of the envelope, there's not much you can do, unless resorting to coupling a noise, or adding a software envelope somewhere to mark the difference.

Fortunately, AT got you covered. The retrig option allow to force the restart of the envelope. There are two ways of using it.

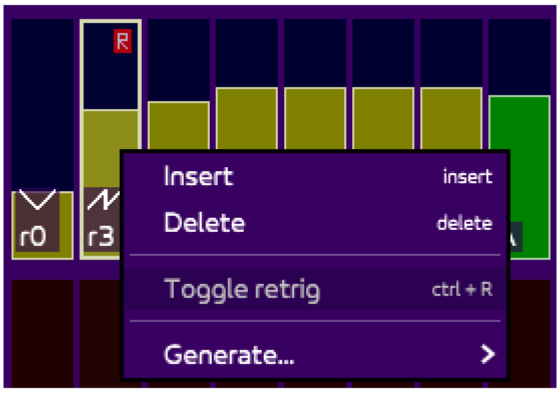
Retrig in the header

On the top of the instrument is this icon: ⊓⊔. What it represents a non-retrig state: the hardware envelope continues where it was, this is the default state. Click on it and it will change to ⊓⊔, meaning that the hardware envelope of the **first column** will be forced for retrig. So it is useless to toggle it unless you have a hardware sound in the first column!

This icon is only a handy shortcut to other technique explained just below.

Retrig in the row

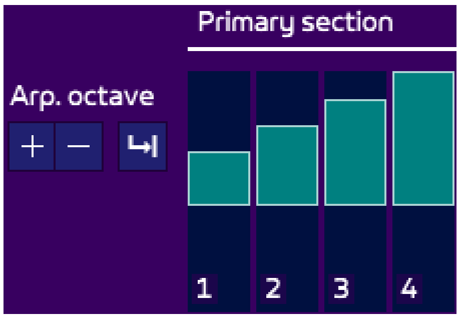
If you set the cursor on an envelope, right-click and select "Toggle retrig", or use ctrl+r, a small red "R" will appear, meaning that this envelope will be forced-triggered.



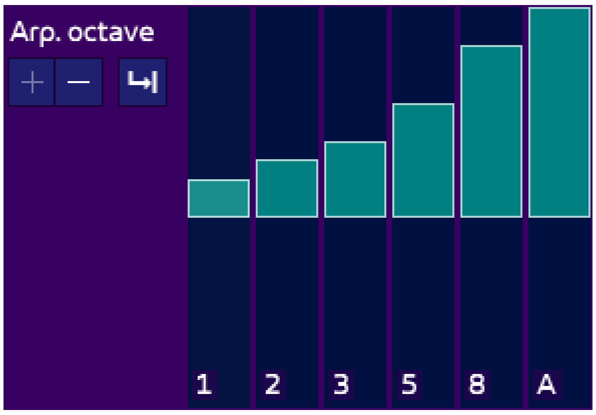
Note that using a retrig in a loop sounds awful most of the times. Consider it a mistake, but maybe you can use this as a special effect (note: Buzz-sync SID, invented by TAO on Atari ST, is actually an effect that relies on this trick, but just like all SID effects, it requires very accurate timing, which current AT players, and AT itself, don't handle).

Showing more values

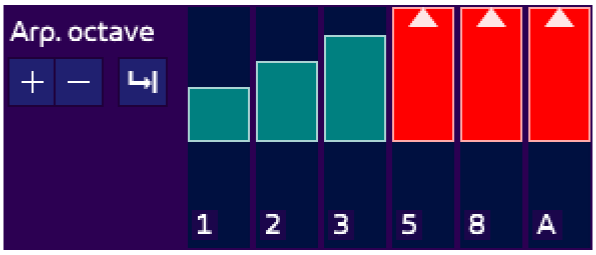
In order to save space and hide useless data to the user, AT only shows the most commonly used values in the rows. For example, the arpeggio octave, by default, shows only from -4 to 4, because it is very rare that one uses more than that. However, you may want to do so. An example:



The area cannot access a value higher than 4. How to get access to all the values? Very simple: use the "+" on the left to increase the area height. You can they add any value you want.



Note that when you get back to a smaller area by pressing the "-" on the left, out-of-bounds values will be shown in red with an arrow on top to highlight the fact that what you see is not the actual values (nothing to be feared actually if you changed something you didn't want to: undo works everywhere!).



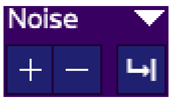
More about that just below!

Hiding and shrinking rows

As you can see, a sound is composed of many rows, but as you will learn, most of the sounds will only use a few of them. No need to display all of them in full! That's where the **hide** and **shrink** options come into play.

Hiding

Hiding is done by clicking on the small downwards arrow in every row, such as here:



By clicking on it, it shows/hides the data of the row. A nice thing is that when hiding a row for a specific instrument, leaving to another instrument and going back will restore its state.

Only the envelope row cannot be hidden, but every sound requires it.

When using the cursor, you can:

- hide the row above the cursor with shift + f
- hide the row below the cursor with control + f

Shrinking/expanding

As explained above, if you use only the first values of a row (for example, pitch +1/-1), it would be a waste of space to show a range going from -FFF to +FFF. In that case, you can change the height of a row:

- expand the row where the cursor is shift + g
- shrink the row where the cursor is control + g

Automatic hiding/shrinking

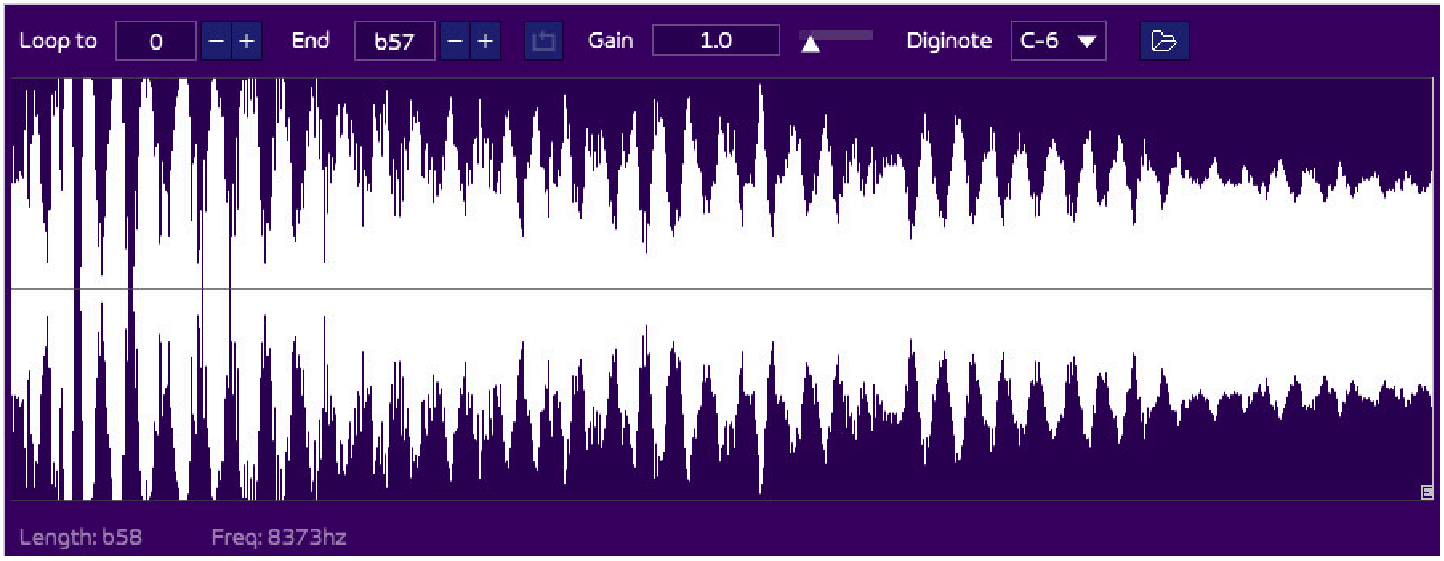
The good news is that all that is explained above can be done automatically, and is automatic on one time: when opening an instrument for the first time, only the rows with significant data are visible. What's more, the rows are shrunk as much as possible to show only the needed values.

However, with time, you may have changed the height of the rows, shown rows which you don't need anymore, etc. Simply use the shrink icon (or use shift + h) to hide all the useless rows and shrink the useful ones!

Some rows can be shrunk even more (the same amount of values, but within a small height), which is handy but reduces a bit the visibility. This is thus not the default option. To shrink even more, right-click on the shrink icon (or use control + h).

The sample instrument editor

If an instrument is a sample, the editor will look like this:



The loop of the sound is shown, and the start/end values can be moved using the mouse, or changed via the LoopTo/End parameters. The loop can be deactivated by unchecking the icon, but the End parameter remains active in this case: it is possible to have a large sample but only plays a subset of it.

The gain is a handy feature: very often, you will notice that our poor dynamic-less hardware struggle to reproduce soft samples. They need a boost. You can do this with your favorite WAV editor, but but tweaking the volume ratio, the volume is changed in real-time. This is a non-destructive operation: the sample is not modified so it’s easy to reduce the volume if the modified value is wrong.

The diginote is an important feature if you use samples in the event tracks. Also, check this page for important technicalities.

For the moment, there is no possibility to edit the sample (cut, trim, etc.). This may be done in the future if requested, but AT3 will never aim at being a real sample editor, knowing that excellent editors already exist.

The arpeggio editor

The Arpeggio Editor (**AE**) allows to create and edit… arpeggios. Arpeggios are considered an "expression" in AT terminology, as opposed to instruments.

What is an arpeggio?

For those who don't know what an arpeggio is, it is simply a loop of several notes (at least 2, but 3 most of the times) that is played generally pretty fast.

Arpeggios can be played with "real" instruments, but they are very popular in sound-chip music because they help circumvent a limitation of the hardware: the channel count.

Imagine you want a chord in your song. On a piano you would play 3 notes at the same time with one hand, and add even more with the other. On a 8-bit, where most hardware have 3 channels at best, playing these 3 notes would leave you with no more channels free to add more arrangements.

Arpeggios allows you to go from this…

0:00 / 0:01

… to this!

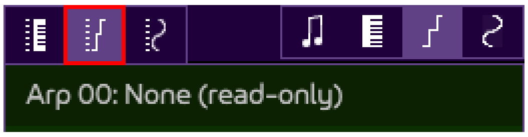
0:00 / 0:01

This sounds different of course, but then there are 2 channels left to add anything you need (bass, melody, drums… each of these instruments can also use arpeggios for an even richer music!).

Arpeggio can be musical, as you've just heard, or completely experimental. They can also have the size you want, so you should not limit yourself to the 3 notes everybody uses.

The arpeggio list

On the left side of the AT3 screen is a list of various items, such as instruments, arpeggios and pitches.



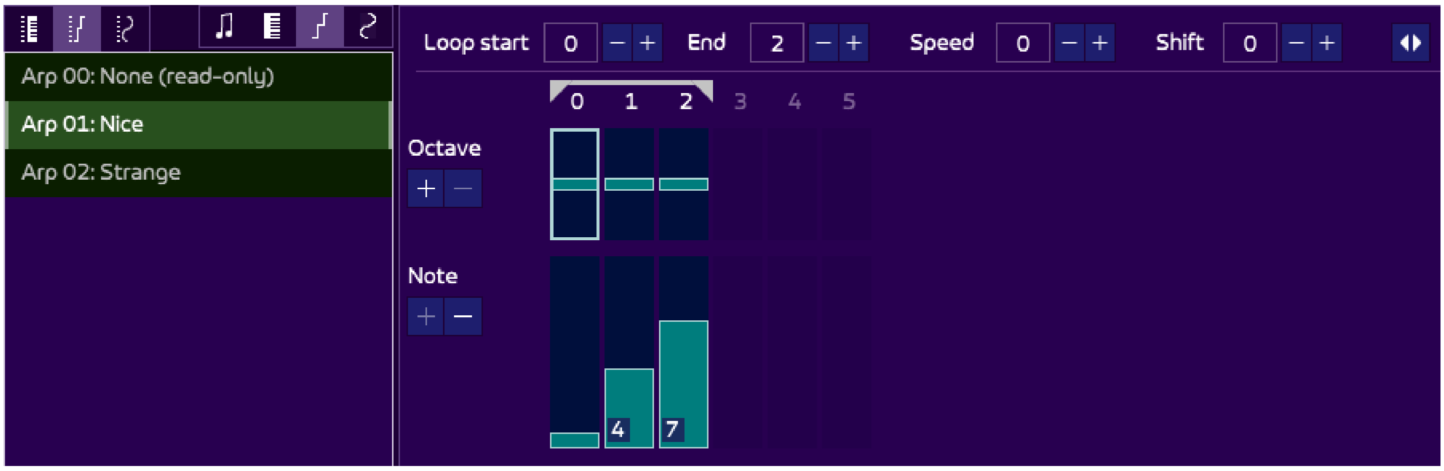
Open the arpeggio list (**AL**) by

- clicking on shift + F4
- or clicking on the second icon of the icons to the left (as highlighted above).

The editor

The AE itself opens either by:

- pressing F4
- clicking on the third icon of the icons to the right
- or by double clicking on an item (such as "Empty" in the song opened when AT3 is launched).



In this screenshot, we have 2 arpeggios: "Nice" (which is shown in the editor on the right), and "Strange". The "None" at the top means "no arpeggio", so we don't really count it as one.

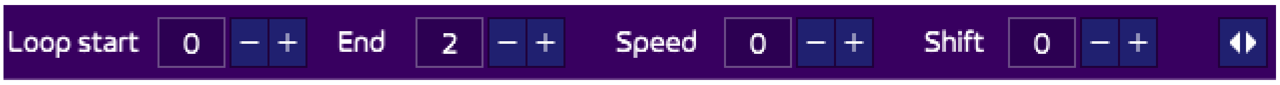
The AE looks much like the instrument editor, which you should be acquainted with by now.

Two rows are present:

- the octave, here with no values, indicating "0".
- the note, here with values 0, 4, 7 (forming a major chord. The "0" is not written, to make the interface less polluted with numbers).

The note indicates what semi-tones to add to a base note (from the music, or your keyboard), to which is added the octave value.

At the top is a header:


Loop start 0 − + End 2 − + Speed 0 − + Shift 0 − + ◀▶

The *loop start* and *end* indicate what part to loop. Note that contrary to instruments, arpeggios **always** loop.

The *speed* goes from 0 (fastest) to 255 (slowest). And just like instruments, you can override this speed in your patterns via the Arpeggio Speed effect, so that you don't have to duplicate an arpeggio just because you want to change its speed occasionally.

Shift is new though. It is only a convenience that allows you to add virtual "0" at the beginning of the arpeggio. The loop is also shifted. It is useful if you want to have your arpeggio to start with a little delay. You can use this parameter instead of adding empty bars by yourself.

Finally, the loop icon ◀▶ changes the horizontal zoom of the bars, which can be useful to visualize the arpeggio better if it grows long.

Editing

The way to edit the bars is exactly the same as for the instrument editor, that is, either via the mouse, the cursor, or by typing them.

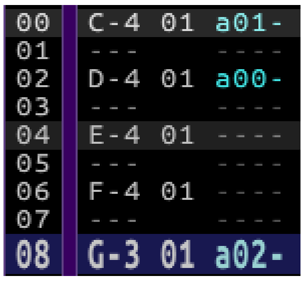
Testing

Just like for the instrument editor, you can use your MIDI keyboard, the QWERTY keys of your computer keyboard, or even the piano at the bottom of the screen.

Using an arpeggio in a song

Once you feel satisfied with an arpeggio, you can use it in song. As you can see in the AL on the left, each arpeggio has a number associated to it (1 for Nice, 2 for Strange). It is this number that you use in the pattern, thanks to the Arpeggio Table effect, represented by the "a" letter.

By writing this little pattern (at slow tempo)…



```
00  C-4 01 a01-
01  ---    ----
02  D-4 01 a00-
03  ---    ----
04  E-4 01 ----
05  ---    ----
06  F-4 01 ----
07  ---    ----
08  G-3 01 a02-
```

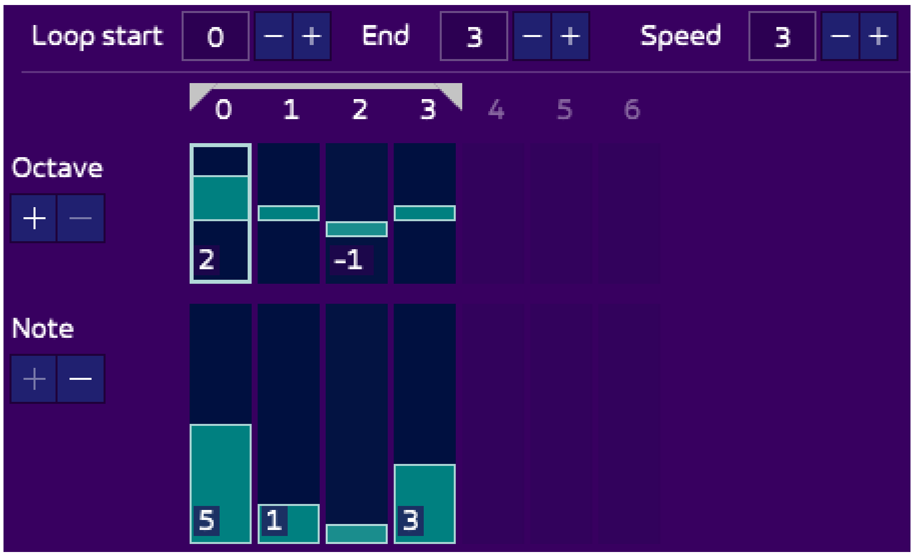
… you can hear this:

0:00 / 0:05

Let’s break it down:

- `C-4 01 a01` means that the arpeggio `01` is used, that is, “Nice”.
- `D-4 01 a00` means that the arpeggio `00` is used. This special value means “no arpeggio”, so any arpeggio played before is stopped. The instrument goes “simple”.
- The two following notes have no effect, so they continue without.
- `G-3 01 a02` triggers another arpeggio, `02` (“Strange”), which sounds just like its name implies!

It looks like this:



There is nothing really musical to it, it was only to show you that you can make strange instruments with arpeggios, sometimes with surprising results. Don’t hesitate to make all tests in the world to discover new sounds!

The pitch editor

The Pitch Editor (**PE**) allows you to create and modify pitches. Pitches are considered an “expression” in AT terminology, as opposed to instruments.

What is a pitch?

If you are a guitar player, you may know it as “pitch bend” or “slide”. In our case, it is also used to simulate a vibrato. A pitch is simply a slight change of frequency, a shift from one note to another. It can be musical or a special effect.

Here is a upward pitch:

0:00 / 0:02

And a nice vibrato:

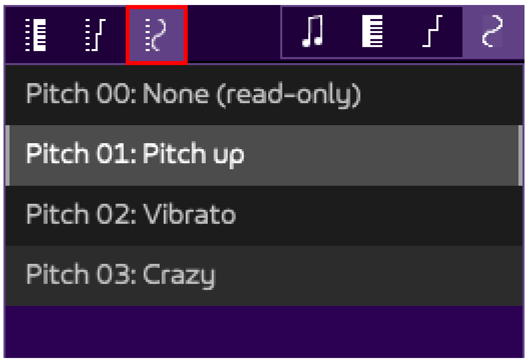
0:00 / 0:02

But you can get crazy and do this:

0:00 / 0:02

The pitch list

On the left side of the screen is a list of various items, such as instruments, arpeggios, and, without surprise, pitches.


Pitch 00: None (read-only)
Pitch 01: Pitch up
Pitch 02: Vibrato
Pitch 03: Crazy

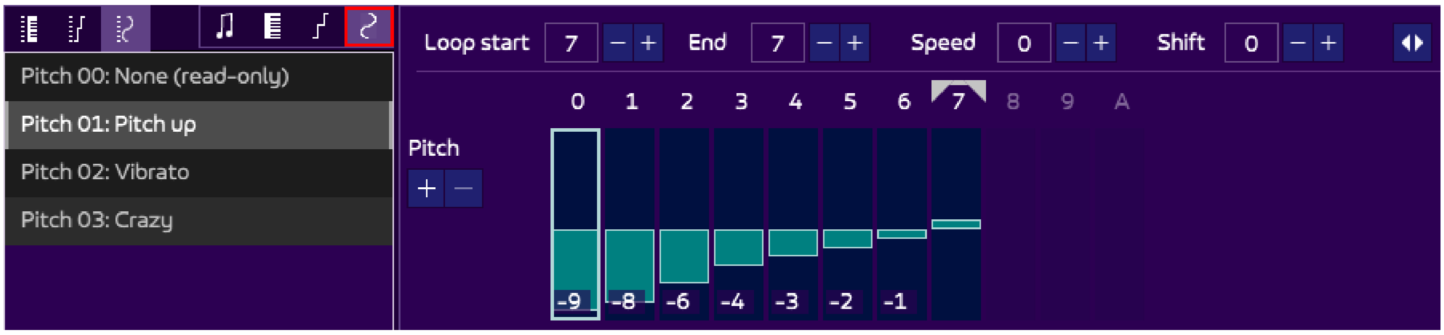
Open the pitch list (**PL**) by

- clicking on shift + F5
- or clicking on the third icon of the icons to the left (as highlighted above).

The editor

The PE itself opens either by:

- pressing F5
- clicking on the fourth icon of the icons to the right
- or by double clicking on an item (such as “Pitch up” in the screenshot above).


Pitch 00: None (read-only)
Pitch 01: Pitch up
Pitch 02: Vibrato
Pitch 03: Crazy

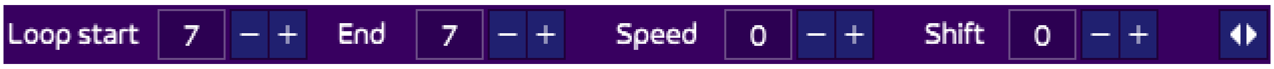
Loop start 7 End 7 Speed 0 Shift 0

In this screenshot, there are 3 pitches : “Pitch up”, “Vibrato” and “Crazy”. The “None” is always present and selecting it means “no pitch”, so doesn’t really count as one.

The PE looks much like the instrument editor, and even more like the arpeggio editor, which both should be known to you by now.

One row only is present, and is the pitch. Each value is added to the note frequency. Negative values will sound lower, positive higher. Use the “+” and “-” icons to increase the visible range of the pitch if you need more of them.

At the top is a header:


Loop start 7 − + End 7 − + Speed 0 − + Shift 0 − +

The *loop start* and *end* indicate what part to loop. Note that contrary to instruments and just like arpeggios, pitches **always** loop.

The *speed* goes from 0 (fastest) to 255 (slowest). And just like instruments, you can override this speed in your patterns via the Pitch Speed effect, so that you don't have to duplicate a pitch just because you want to change its speed occasionally.

Shift is new though. It is only a convenience that allows you to add virtual "0" at the beginning of the pitch. The loop is also shifted. It is useful if you want to have your pitch to start with a little delay. You can use this parameter instead of adding empty bars by yourself. Typically, you may want a vibrato to start a bit later than the start of the note.

Finally, the loop icon changes the horizontal zoom of the bars, which can be useful to visualize the pitch better if it grows long.

Editing

The way to edit the bars is exactly the same as for the instrument editor, that is, either via the mouse, the cursor, or by typing them.

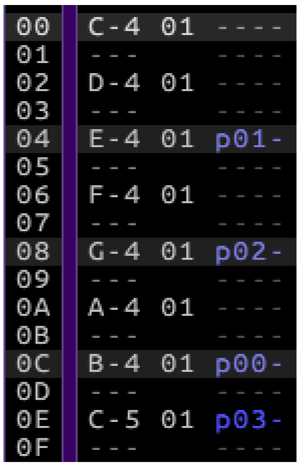
Testing

Just like for the instrument editor, you can use your MIDI keyboard, the QWERTY keys of your computer keyboard, or even the piano at the bottom of the screen.

Using a pitch in a song

Once you feel satisfied with a pitch, you can use it in song. As you can see in the PL on the left, each pitch has a number associated to it (1 for "Pitch up", 2 for "Vibrato", etc.). It is this number that you use in the pattern, thanks to the Pitch Table effect, represented by the "p" letter.

By writing this little pattern (at slow tempo)…



```
00  C-4 01 ----
01  ---    ----
02  D-4 01 ----
03  ---    ----
04  E-4 01 p01-
05  ---    ----
06  F-4 01 ----
07  ---    ----
08  G-4 01 p02-
09  ---    ----
0A  A-4 01 ----
0B  ---    ----
0C  B-4 01 p00-
0D  ---    ----
0E  C-5 01 p03-
0F  ---    ----
```

… you could hear this:

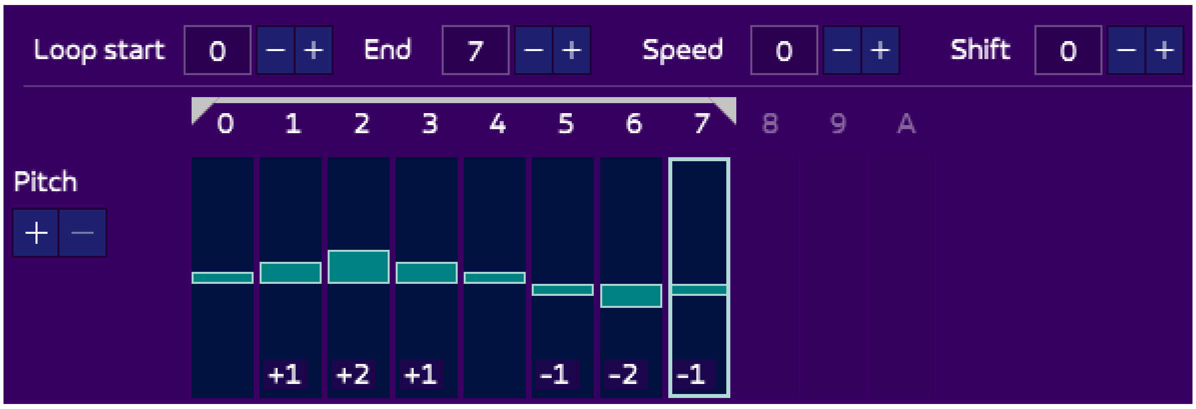
0:00 / 0:06

Let's break it down:

- The first two notes (`C-4 01` and `D-4 01`) have no effect.
- `E-4 01 p01` means that the pitch `01` is used ("Pitch up").
- `F-4 01` has no effect. However, pitches, just like arpeggios, start again at each note, so the latest pitch applies. There is no need to continuously write it!
- `G-4 01 p02` plays the pitch `02` ("Vibratos"), which you can also hear on the next note, `A-4 01`.
- `B-4 01 p00` has a special meaning: by using pitch `00`, we stop the possible pitch.
- `C-5 01 p03` triggers pitch `03` ("Crazy"), which sounds just like its name implies!

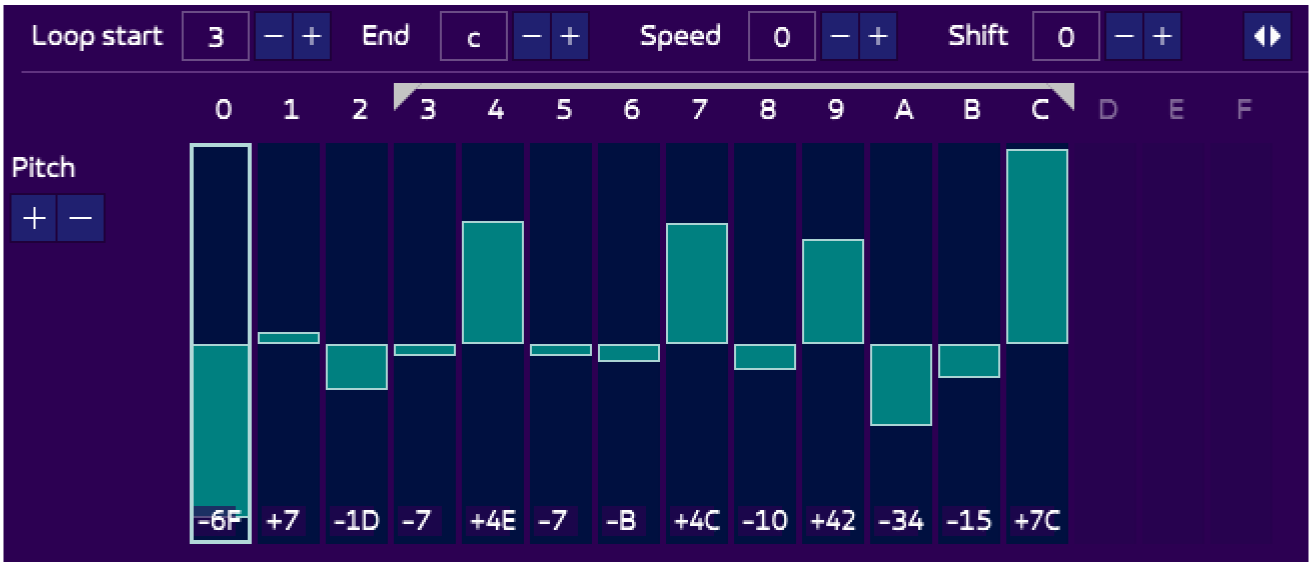
The "Pitch up" pitch can be seen in the screenshot above.

"Vibrato" looks like this:


Loop start 0 End 7 Speed 0 Shift 0

Pitch: 0, +1, +2, +1, 0, -1, -2, -1

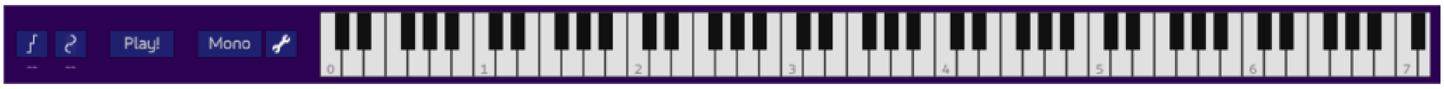
"Crazy" looks like this:


Loop start
3
End
c
Speed
0
Shift
0
Pitch
0 1 2 3 4 5 6 7 8 9 A B C D E F
-6F +7 -1D -7 +4E -7 -B +4C -10 +42 -34 -15 +7C

The nice thing about Expressions is that they stack up: you can have an arpeggio and a pitch at the same time, for very expressive (and sometimes unexpected) results.

The test area

At the bottom is the test area (**TA**). It allows you to play with your instruments/expression:



By clicking on the keyboard notes, the currently selected instrument (in the Instrument List at the left of the screen) plays.

When focused, you can also use the computer keyboard to play notes (qwerty).

There are several icons:

Use Arpeggio on/off

When the toggle is on ( ) the instrument being played will use the selected arpeggio (which you can change in the Arpeggio list).

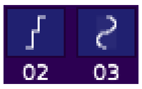
When off ( ) no arpeggio is use.

Use Pitch on/off

The same as the arpeggio, but with the pitch (on or off).

Used Arpeggio/pitch number

Below the two icons on the left are "–". This is when no arpeggio/pitch is selected. However, when you select either in the Arpeggio List/Pitch List, their number is shown. This is useful for you to understand why the sound you play sounds like it does: a vibrato may be heard, a major arpeggio, etc:



Play note

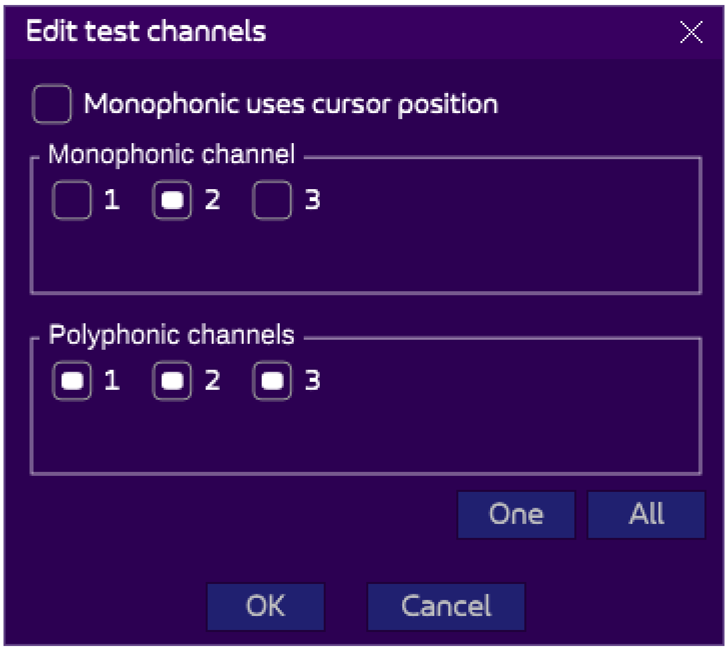
This button Play!, when clicked, will play a note, which is displayed (C-4 for example).

Monophony/polyphony

This toggles between monophonic Mono and polyphonic Poly behavior (detailed set-up is below). Monophonic will play the note on only one channel. Polyphonic will play every note on the next channel.

Setting up the mono/polyphony

This icon opens a dialog to set up how the mono/polyphony behaves:



In the monophonic section, you can select one channel where you want the instrument played. Selecting the second is the usual choice, as it is in the middle channel. The “monophonic uses cursor position” toggle, when on, will disable the monophonic selection and automatically select the same channel as where the cursor is in the Pattern Viewer.

In the polyphonic section, you can select all channels on which the instrument (with the computer keyboard) plays, one after the one. In the example of the screenshot, playing notes will alternate between channel 1, 2, 3 before going back to channel 1.

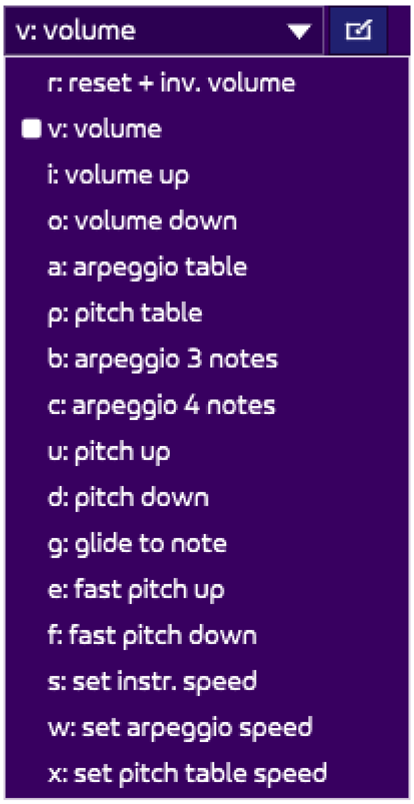
Test Area/Pattern Viewer behavior

Important: when the Pattern Viewer (**PV**) is focused, playing a note is using the “PV behavior”: that is, the effects that are currently used by the PV (volume, arpeggio, pitch, etc.), and using the channel where the cursor is. However, when any **other panel** is focused, playing a note uses the “Test Area behavior”: the volume is full, the arpeggio and pitch use are the selected ones, if their related icons at the bottom-left are on. The monophonic/polyphonic settings is also used.

Check the effect context page for a nice schema.

Effects

Arkos Tracker has various effects to be added to your tracks. Please check the Pattern Viewer page for how to write them.



An effect comes with a value of 1, 2 or 3 digits, thus making a value from 0 to F, FF or FFF in hex.

Each effect is represented by a letter (v for volume, etc.). However, these letters all can be customized (File > Setup > Pattern Viewer) if you don't like these, or are used to another tracker effect naming. Also note that all the effects have a little description on the top-right of the Pattern Viewer when you put the cursor over an effect.

Here is a list of all the effects and their use.

Volume (vX–)

This sets the volume of the current track, from F (full volume) to 0 (mute). This is not an amplification: using F means the volumes of the instruments are used, without modification.

Volume effects can be used along with a note, or besides.

You can stack up volume effects to create, for example, fade-outs (vF, vE, vD, vC, … v0) but this is not optimal, though this was how you did it in STarkos, Arkos Tracker 1, or many old-school format like MOD. It is much more flexible to set the volume (vF for example), and then use the volume up/down effects (see below) for a smooth decay. Less typing, more accuracy!

A volume is used by the whole channel, not a specific instrument. So setting a volume to 5 (a rather low value) will result in a very quiet track, regardless of any change of instrument, and will continue on all the next tracks, till another volume (or reset) effect is produced.

Please note that the volume curve on the AY/YM is logarithmic: fade outs/ins will sound pretty “steppy” on the highest volumes.

Also, bear in mind that **hardware sounds cannot be amplified/soften by any effect**. The hardware envelope generates the sound, which varies between a volume of 0 and F, and nothing else.

Volume up/down (iXXX / oXXX)

This makes the volume automatically go… up (“i” for “in”) or down (“o” for “out”) with one command. This is perfect for a fade in/out, or to create a release of a sound.

Note that contrary to many (user-unfriendly) trackers, you don’t need to multiply the command on several lines for the volume slide to continue. Only write it once, and the slide will be performed till the end of time… or another volume effect, or another note is encountered.

As for the XXX, it sets the speed at which the slide is performed. “100” means that every frame, the volume is increased or decreased of one, which would be pretty fast, as the volume is between 0 and F. However, the two last digits allow a very accurate slide: “080” is twice slower than “100”, and “001” would mean increasing the volume of 1 every 256 frame!

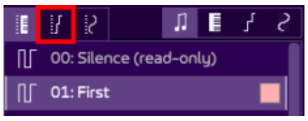
To stop the volume slide, you can use a speed of “000”.

Arpeggio table (aXX)

This effect allows to start an Arpeggio, with “XX” indicating what table to use, from “01” to “FF”.

Arpeggio effect continue even the another note, even using another instrument, is used. You can stop the arpeggio with the value “00”, or with the Reset effect (but other effects would be reset as well).

Note that arpeggios are restarted on every new note.

The arpeggio tables can be created with the arpeggio editor, which you can open via the list on the left: 

Pitch table (pXX)

Starts a Pitch, with “XX” indicating what table to use, from “01” to “FF”. Just like the arpeggio effect above, the pitch can be stopped with the value “00”, and will continue even if a new note or new instrument is used. And likewise, the pitch tables can be created with the pitch editor via the list on the left.

Arpeggio 3 notes (bXX)

This is very much like the Arpeggio Table effect above, but instead of having to create a table, you can “*inline*” the value in the pattern. For example b37 will play a minor chord (base note, a third, and a seventh).

You can stop the effect by using “b00”, or “a00”, or even “c000” (see below), that is, any arpeggio effect.

Why use Arpeggio Table, when inline arpeggio can be used?

- Inline is only limited to 2 additional notes
- Ranging from 0 to F semi-tones (no negative values)
- A default speed of 0 (the fastest. You can override this with the “set arpeggio speed” effect below).
- A loop forced on these 3 notes. You cannot loop on the before-last or last semi-tone.
- On a technical low-level, most exported format (AKG, AKM) will convert inline arpeggios to arpeggio tables, which are limited to 255 at best. If you create many *different* inline arpeggios, the export will fail (don’t worry, I never saw anyone reaching this limitation!).

So inline arpeggios are handy for quick “arpegging”, but are somewhat limited.

Arpeggio 4 notes (cXXX)

The same as “Arpeggio 3 notes”, but with… 4. Now you can have 4-note arpeggios for even more possibilities (yet, always less than with an Arpeggio Table).

Pitch up/down (uXXX / dXXX)

These effects change the frequency of the sound (it goes higher, or lower) at the “XXX” speed. “100” means a period of 1 each frame (pretty fast), so use the two-last digits for a slower result (“020” for example).

Once started, the pitch continues till another note is found, so there is no need spreading it across multiple lines. Use “u000” or “d000” to stop the pitch.

Please note that pitch and arpeggio do not mix well together, musically speaking. It will sound funny and many many songs use that combination, but may not sound as musical as you’d like.

Fast pitch up/down (eXXX / fXXX)

This is exactly like pitch up/down, but… faster. Pitch effects are very accurate, sometimes making specific pitches too slow. Fast pitch is 4 times faster.

Glide to note (gXXX)

This is a variation of the Pitch effect, which allows you to slide a note into another one, at a desired speed. For example, you write the note “A-3”. Several lines below, type “`B-4 g100`“. This will make a slide from A-3 to B-4 with a period of 1 per frame, and the slide will stop when B-4 is reached.

Set instrument speed (sXX)

Each instrument has its own speed. Sometimes, you want to use an instrument you’ve made, but faster or slower. Instead of having to duplicate it to change its speed (which would be inefficient), simply use this effect and here you go. The speed, just like the instrument, goes from 0 (fastest) to FF (slowest).

Set arpeggio speed (wXX)

This is the same as the instrument speed, but for arpeggio. Note that you can also override the “Arpeggio 3 or 4 notes” effects (i.e. inline arpeggio) 0-by-default speed with this (just put it after the inline arpeggio effect).

Set pitch table speed (xXX)

This is the same as the arpeggio speed, but for Pitch Table.

Effect validity

- As the Pattern Viewer will show you, illogical use of effects will mark them in red. Put your cursor on it and read the message to understand what is going on. Mostly, duplicate effects are not authorized. A Volume Up/Down before a Set Volume is useless (the first being overriden by the later), and so on.
- All effects are “normalized”, which mean that they are re-ordered on export, and encoded as “block effect”, which are re-used throughout the song. This means you don’t have to worry about the location of the effects on the line if you thought about optimizing your song.

- Inline arpeggios (b, c effects) are internally converted to Arpeggio Table, which are still limited to 255. If the limitation is broken, the export fails.

Performance

Using effect is CPU consuming. A few things to know if you want to save some CPU, and possibly memory (especially useful for non-streamed players like AKG and AKM):

- A Reset effect costs a bit, because it resets the volume, the pitch, the arpeggio. However, it is better to use it than all three effects which, in sum, cost more.
- Don’t over-use the “set speed” effects for instruments, arpeggio and pitch, as they cost both CPU and memory. If using them *a lot* with always the same value, it might be simpler to duplicate the item with another speed.
- Pitch Glide and Volume In/Out are more CPU/memory consuming than simple Pitch Up/Down and Set Volume. Remember that the player configuration feature can optimize the code that is not useful. If in dire need of CPU/memory, don’t use the aforementioned effects.

The effect context

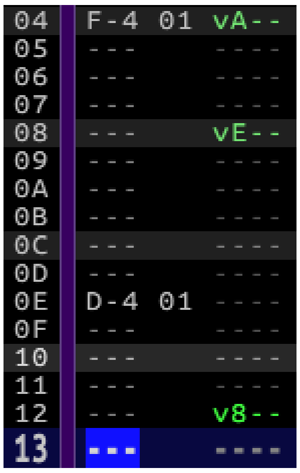
The effect context (EC) is a new feature since 3.2.4, and is quite unique in the world of old-school (or new!) trackers.

What is effect context?

You know that most of the effects in AT have not a local-only action: if you set a volume, it will last for as long as no other volume-related effect is set again. The same for arpeggio and pitch tables: you could trigger an expression and it could remain for the rest of song.

This is a desired behavior. However, when composing and editing your song, you may have fallen into an unwanted case: what you hear isn't what would be heard if you'd play the song or the pattern from the start. That is because the software does not "browse back" the possible effects to apply them in the currently edited location.

An example:



```
04  F-4 01 vA--
05  ---    ----
06  ---    ----
07  ---    ----
08  ---    vE--
09  ---    ----
0A  ---    ----
0B  ---    ----
0C  ---    ----
0D  ---    ----
0E  D-4 01 ----
0F  ---    ----
10  ---    ----
11  ---    ----
12  ---    v8--
13  ---    ----
```

The volume varies at different lines of the track. Logically, at line 13 where the cursor is, the volume should be 8. However, in most tracker, if you'd stop playing, and try to play notes anywhere where no volume effect is, the volume used would be 15, the default value.

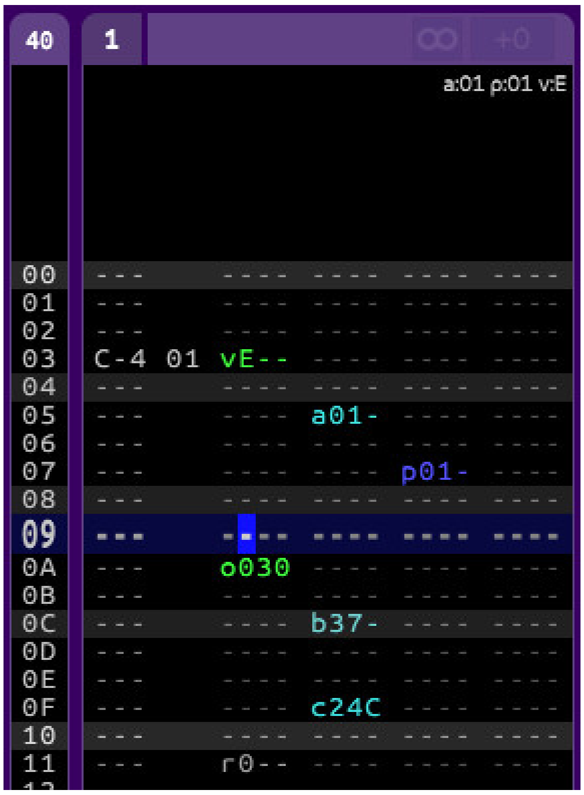
If you'd go to line 12, press Enter to play the line, the volume 8 would be captured. But if you'd go upwards and play the line E, you'd expect to hear it with a volume of E, since the continuous volume of line 8 should be applied below! Well that's *was* not the case… until now.

The effect context in action

EC will allow you to have the continuous effect such as Volume, Arpeggio and Pitch Table to be applied wherever you:

- play a note in the Pattern Viewer
- enter a note in the Pattern Viewer
- play a part of the song (whole song, track, block): it is applied at the beginning of the section, and when it loops.

This might seem complicated, but it is not! EC comes with a visual aid that you will soon learn to love and won't be able to live without. At the top-right of the track where the cursor is, are written the current volume, arpeggio and pitch table. Here an example:

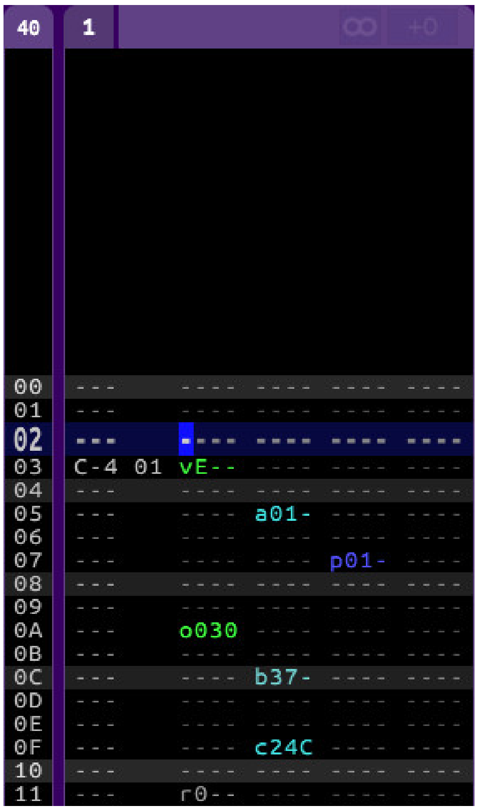


```
                          a:01 p:01 v:E
00  ---       ----  ----  ----  ----
01  ---       ----  ----  ----  ----
02  ---       ----  ----  ----  ----
03  C-4  01   vE--  ----  ----  ----
04  ---       ----  ----  ----  ----
05  ---       ----  a01-  ----  ----
06  ---       ----  ----  ----  ----
07  ---       ----  ----  p01-  ----
08  ---       ----  ----  ----  ----
09  ---       ----  ----  ----  ----
0A  ---       o030  ----  ----  ----
0B  ---       ----  ----  ----  ----
0C  ---       ----  b37-  ----  ----
0D  ---       ----  ----  ----  ----
0E  ---       ----  ----  ----  ----
0F  ---       ----  c24C  ----  ----
10  ---       ----  ----  ----  ----
11  ---       r0--  ----  ----  ----
```

The cursor is at line 9. Above that, the volume is E, the arpeggio table set to 1, the same for the pitch table. Lo and behold, this is also written at the top-right! "a" indicates the arpeggio table, "p" the pitch table, and "v" the volume.

To avoid polluting the interface, only the track where the cursor is will have this visual aid.

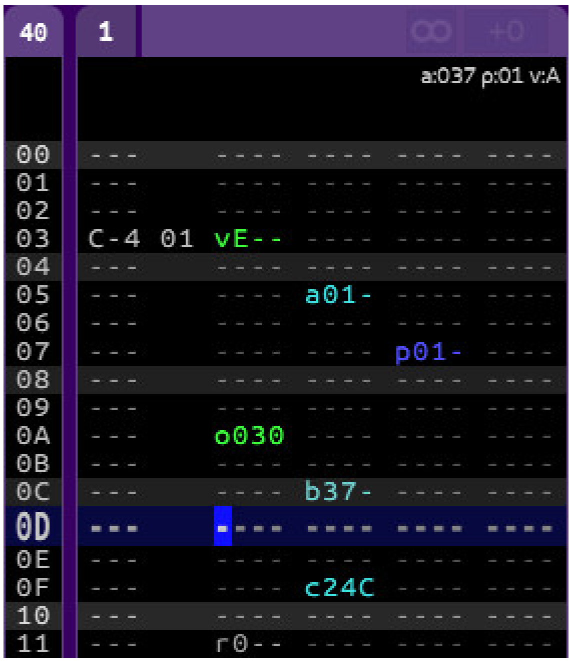
If you move your cursor location, the help is updated accordingly: let's move at the top!



```
40  1                            ∞   +0

00  ---      ----  ----  ----  ----
01  ---      ----  ----  ----  ----
02  ---      ----  ----  ----  ----
03  C-4 01   vE--  ----  ----  ----
04  ---      ----  ----  ----  ----
05  ---      ----  a01-  ----  ----
06  ---      ----  ----  ----  ----
07  ---      ----  ----  p01-  ----
08  ---      ----  ----  ----  ----
09  ---      ----  ----  ----  ----
0A  ---      o030  ----  ----  ----
0B  ---      ----  ----  ----  ----
0C  ---      ----  b37-  ----  ----
0D  ---      ----  ----  ----  ----
0E  ---      ----  ----  ----  ----
0F  ---      ----  c24C  ----  ----
10  ---      ----  ----  ----  ----
11  ---      r0--  ----  ----  ----
```

Here on line 2, no effect has been triggered yet, so nothing is written at the top-right. Yet you may wonder, the volume is F, why isn't it written? Simply because it is the default volume, so it is not worth being written.

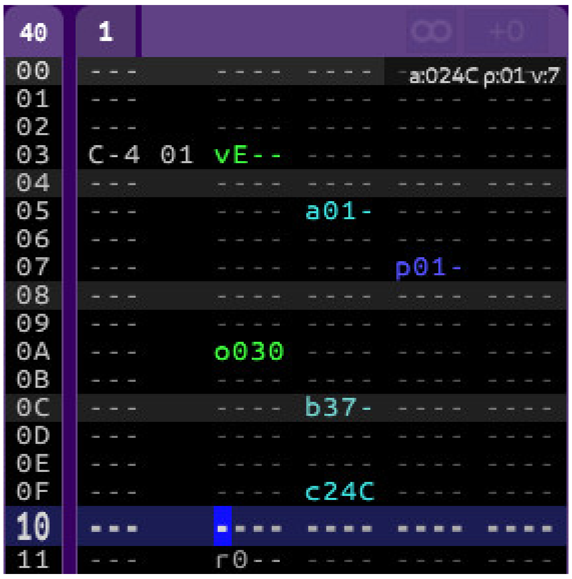
Let's move below:



```
40  1                            ∞   +0
                          a:037 p:01 v:A

00  ---      ----  ----  ----  ----
01  ---      ----  ----  ----  ----
02  ---      ----  ----  ----  ----
03  C-4 01   vE--  ----  ----  ----
04  ---      ----  ----  ----  ----
05  ---      ----  a01-  ----  ----
06  ---      ----  ----  ----  ----
07  ---      ----  ----  p01-  ----
08  ---      ----  ----  ----  ----
09  ---      ----  ----  ----  ----
0A  ---      o030  ----  ----  ----
0B  ---      ----  ----  ----  ----
0C  ---      ----  b37-  ----  ----
0D  ---      ----  ----  ----  ----
0E  ---      ----  ----  ----  ----
0F  ---      ----  c24C  ----  ----
10  ---      ----  ----  ----  ----
11  ---      r0--  ----  ----  ----
```

Here, some new effects have been added: the "o" volume will decrease the volume as times evolves. At line D, it has decreased to A, as written at the top-left! What about the arpeggio? On line C, an inline arpeggio is used: `a:037` is written in the EC helper. It is still an **A**rpeggio, composed of the relatives notes `+0 +3 +7`.

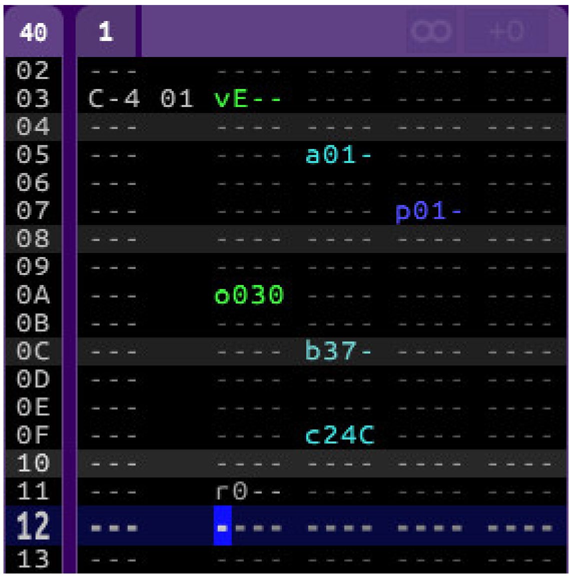
Moving down:



```
40   1
00   ---     ---- ----   a:024C p:01 v:7
01   ---     ---- ---- ---- ----
02   ---     ---- ---- ---- ----
03   C-4 01  vE-- ---- ---- ----
04   ---     ---- ---- ---- ----
05   ---     ---- a01- ---- ----
06   ---     ---- ---- ---- ----
07   ---     ---- ---- p01- ----
08   ---     ---- ---- ---- ----
09   ---     ---- ---- ---- ----
0A   ---     o030 ---- ---- ----
0B   ---     ---- ---- ---- ----
0C   ---     ---- b37- ---- ----
0D   ---     ---- ---- ---- ----
0E   ---     ---- ---- ---- ----
0F   ---     ---- c24C ---- ----
10   ---     ---- ---- ---- ----
11   ---     r0-- ---- ---- ----
```

With our cursor at line 10, the arpeggio effect has been replaced by a four-note arpeggio (effect C), so `a:024C` is written at the top-right. Notice that the volume has continued to decrease, and has now reached 7!

What happens after the Reset at the bottom?



```
40   1
02   ---     ---- ---- ---- ----
03   C-4 01  vE-- ---- ---- ----
04   ---     ---- ---- ---- ----
05   ---     ---- a01- ---- ----
06   ---     ---- ---- ---- ----
07   ---     ---- ---- p01- ----
08   ---     ---- ---- ---- ----
09   ---     ---- ---- ---- ----
0A   ---     o030 ---- ---- ----
0B   ---     ---- ---- ---- ----
0C   ---     ---- b37- ---- ----
0D   ---     ---- ---- ---- ----
0E   ---     ---- ---- ---- ----
0F   ---     ---- c24C ---- ----
10   ---     ---- ---- ---- ----
11   ---     r0-- ---- ---- ----
12   ---     ---- ---- ---- ----
13   ---     ---- ---- ---- ----
```

Well, since the Reset has restored the arpeggio/pitch/volume to their default values, it is no surprise that the helper does not show any value.

Of course, the power of the Effect Context will also apply seamlessly across positions! One volume could decrease slowly during several positions, the EC would still “follow” it. All the same, an arpeggio triggered at the first position would continue up to any following position, until it is changed or reset.

Looping a song

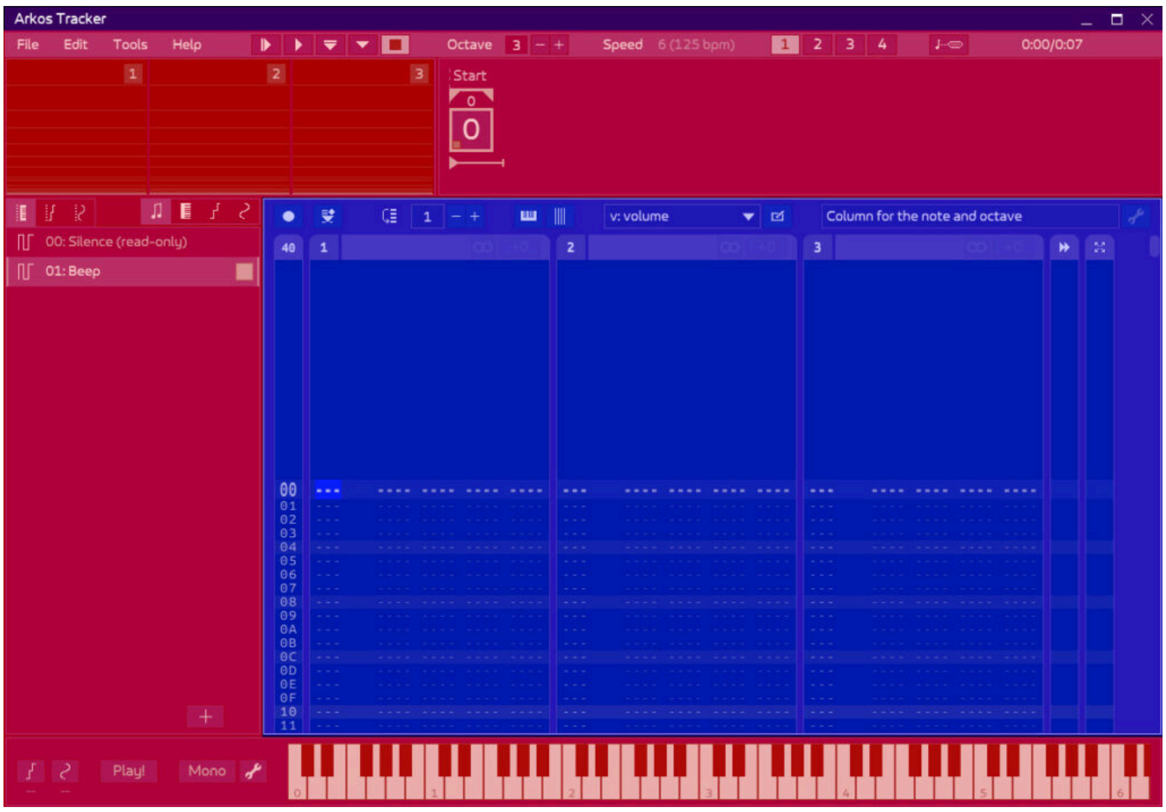
You may wonder what happens when the song loops. Indeed, many times sounds or effects that are triggered at the end of your song will be heard at the beginning of the song, if you didn't explicitly reset them. Sometimes on purpose, sometimes by mistake (in which case you will correct your song by adding a Reset/Volume/Arpeggio/Pitch in the first position).

But how will the Effect Context react to the loop? Indeed, when playing a block or a pattern, we *want* the EC to be applied at its beginning.

This is exactly what AT does: when playing a song, from the beginning or not, the **loop** will **not** trigger the EC, which is what you'd hear on the hardware with the (non-streamed) players. However, when playing anything else (a pattern, a block, a line), EC **will** be triggered when looping, which is very handy for editing smaller chunks of your song.

Effect context area

Effect Context **does not apply to any other panel** than the Pattern Viewer, since the "Test Area behavior" would apply in such areas. Here is a little schema to help you understand:



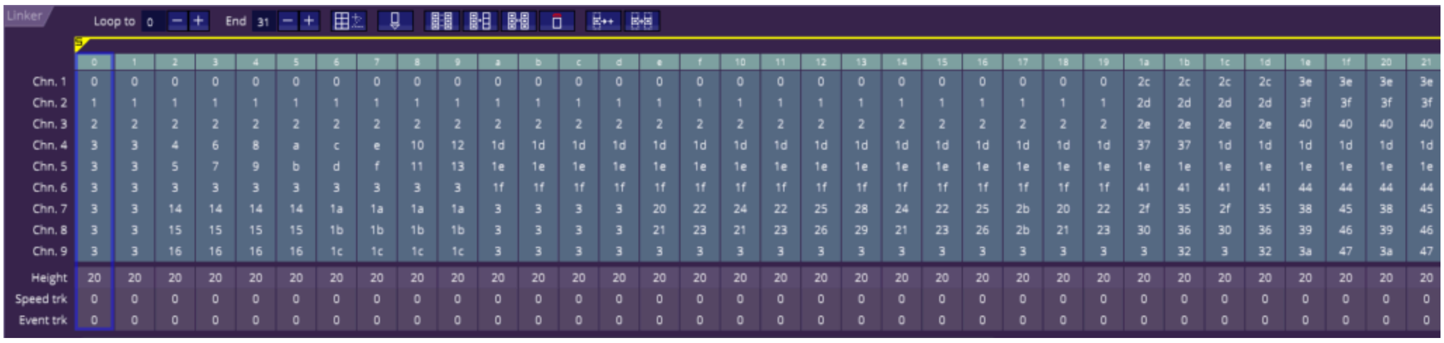
- If playing a note when the blue panel (i.e. the PV) is focused, you are using the "PV behavior", and thus uses the Effect Context. So the volume, pitch and arpeggio are applied according to where your cursor is.

- If playing a note in any other panel (in the red area, such as Linker, the List on the left, or the Test Area at the bottom), it is played according to the “Test Area behavior”, that is, using a full volume and the arpeggio/pitch selected in the expression lists.

Linking

Linking is an advanced feature, so you may want to keep this for later if you're new to AT3.

If you have already used STarKos, AT1 or AT2, you must know that all positions had tracks which were directly referred to via a track number. This was, to say the least, rather clumsy. Remember this?



Linker

Loop to 0 End 31

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Chn. 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Chn. 2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Chn. 3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Chn. 4	3	3	4	6	8	a	c	e	10	12	1d	1d	1d	1d	1d	1d
Chn. 5	3	3	5	7	9	b	d	f	11	13	1e	1e	1e	1e	1e	1e
Chn. 6	3	3	3	3	3	3	3	3	3	3	1f	1f	1f	1f	1f	1f
Chn. 7	3	3	14	14	14	14	1a	1a	1a	1a	3	3	3	3	20	22
Chn. 8	3	3	15	15	15	15	1b	1b	1b	1b	3	3	3	3	21	23
Chn. 9	3	3	16	16	16	16	1c	1c	1c	1c	3	3	3	3	3	3
Height	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
Speed trk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Event trk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f	20	21
Chn. 1	0	0	0	0	0	0	0	0	0	0	2c	2c	2c	2c	3e	3e	3e	3e
Chn. 2	1	1	1	1	1	1	1	1	1	1	2d	2d	2d	2d	3f	3f	3f	3f
Chn. 3	2	2	2	2	2	2	2	2	2	2	2e	2e	2e	2e	40	40	40	40
Chn. 4	1d	1d	1d	1d	1d	1d	1d	1d	1d	1d	37	37	1d	1d	1d	1d	1d	1d
Chn. 5	1e	1e	1e	1e	1e	1e	1e	1e	1e	1e	1e	1e	1e	1e	1e	1e	1e	1e
Chn. 6	1f	1f	1f	1f	1f	1f	1f	1f	1f	1f	41	41	41	41	44	44	44	44
Chn. 7	24	22	25	28	24	22	25	2b	20	22	2f	35	2f	35	38	45	38	45
Chn. 8	21	23	26	29	21	23	26	2b	21	23	30	36	30	36	39	46	39	46
Chn. 9	3	3	3	3	3	3	3	3	3	3	3	32	3	32	3a	47	3a	47
Height	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
Speed trk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Event trk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Yup, this was AT2 with a 9-channel song. Not a pretty sight. Not only you had to remember what was the number of the tracks you wanted to re-use, but you could accidentally modify a track used several times without wanting to.

AT3 linker makes it simpler and manages duplicated tracks by itself. However… sometimes the possibly to have several tracks *linked* together was useful:

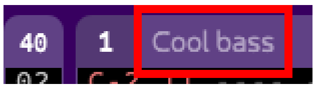
- You could work on a track that you know will sound the same somewhere else, so modifying it in one place to see it modified everywhere else is nice.
- You might want to fine-tune a music for a production with a heavy memory constraint (a 4k demo for example), and here again, ensuring that parts of the songs are the same would facilitate the optimization.

In both these cases, *linking* is what you need.

Linking will allow you to share a track among different patterns. Modifying the track in one of these patterns will have it modified in the other patterns too.

How to link?

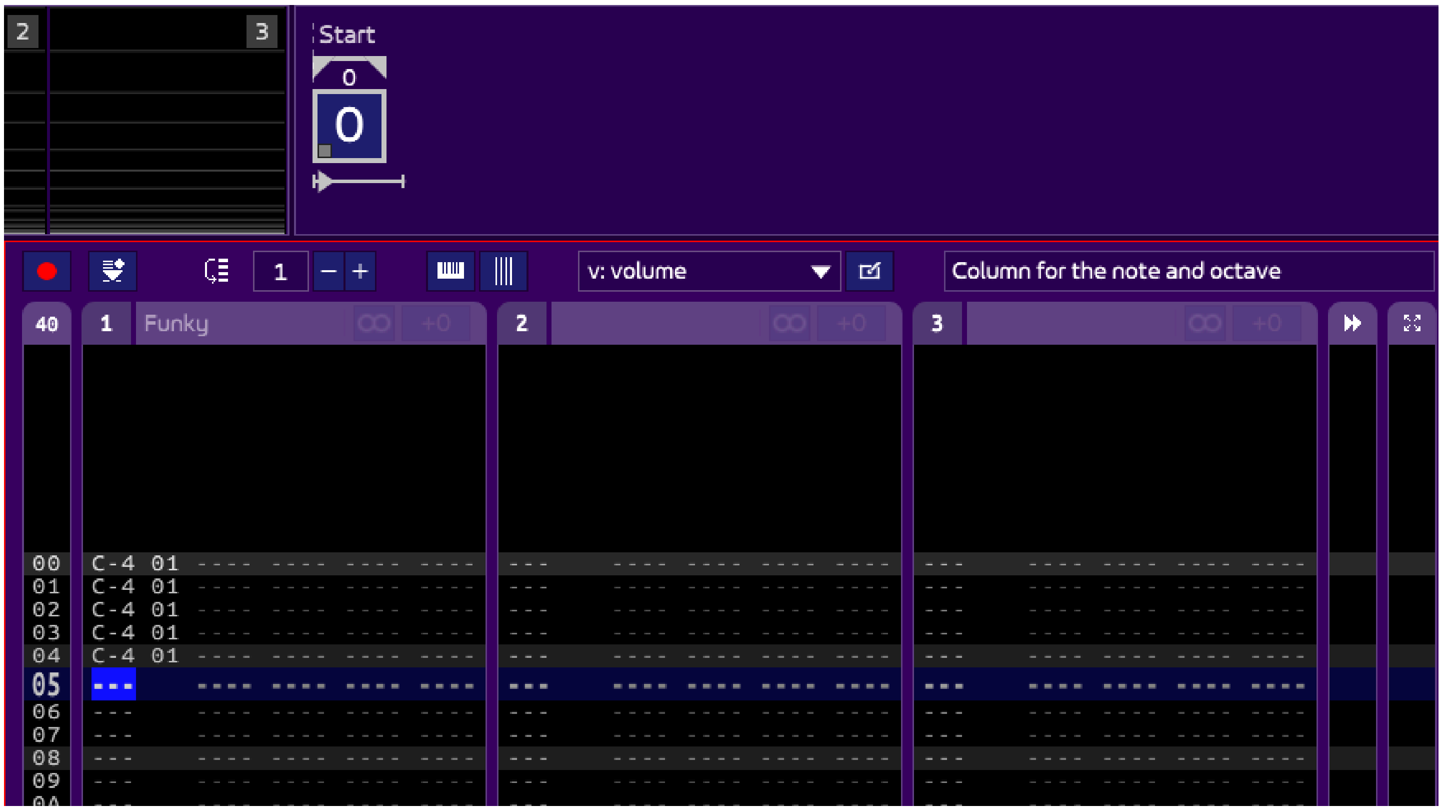
Linking was done via clumsy track numbers in previous software. AT3 does it elegantly via *track names*. As you probably have seen in the pattern viewer, you can name tracks in every pattern by clicking on its header (this also includes the speed/event tracks):



Now that you've named a track, **other tracks** can link to it. Track names replace track numbers! Here is an example.

- Creates a new empty song.
- Enters a few notes in the first channel of the first position.
- Name this track "Funky".

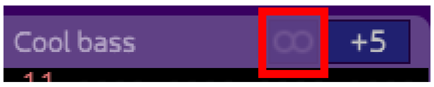
This could look like this:



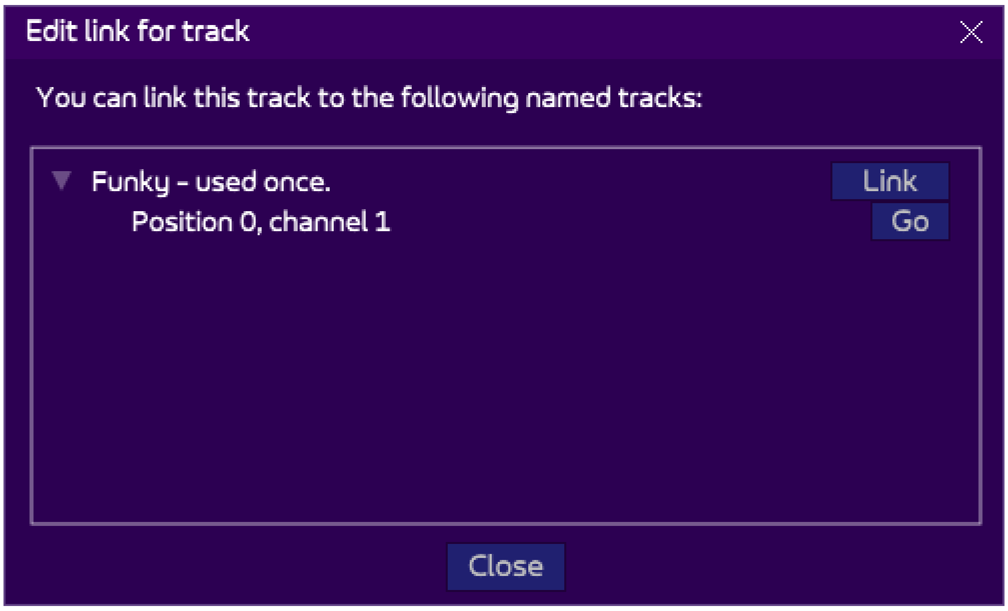
```
00 C-4 01
01 C-4 01
02 C-4 01
03 C-4 01
04 C-4 01
05
```

Now imagine you want another track to re-use "Funky", and most of all, that modifying "Funky" in one place also modifies it on the other. For simplicity's sake, we will make this in the same pattern.

Notice the almost invisible "chain" icon on every track:



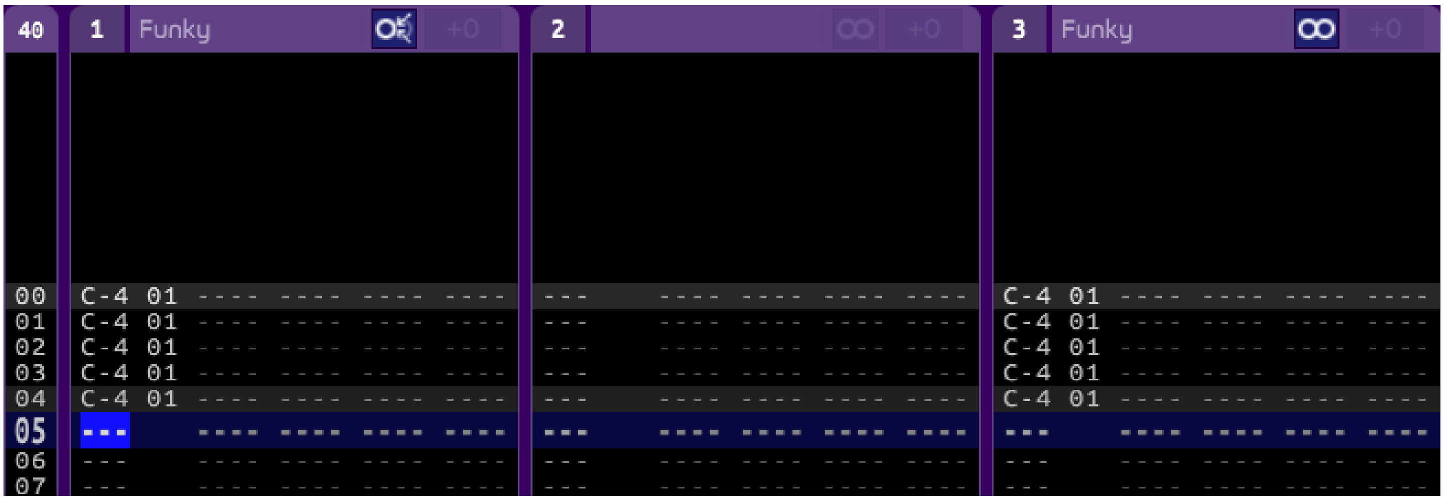
Click on the one of the third channel. It will open a dialog (here with the tree expanded):


Edit link for track

You can link this track to the following named tracks:

Funky - used once.
Position 0, channel 1

This indicates that you can *link* the track of the third channel *to* “Funky”, the track of the first channel. If in the future, you’re not sure about what this track is, you can click on “Go” to go listen to it. For now, click on “Link”. And watch this!

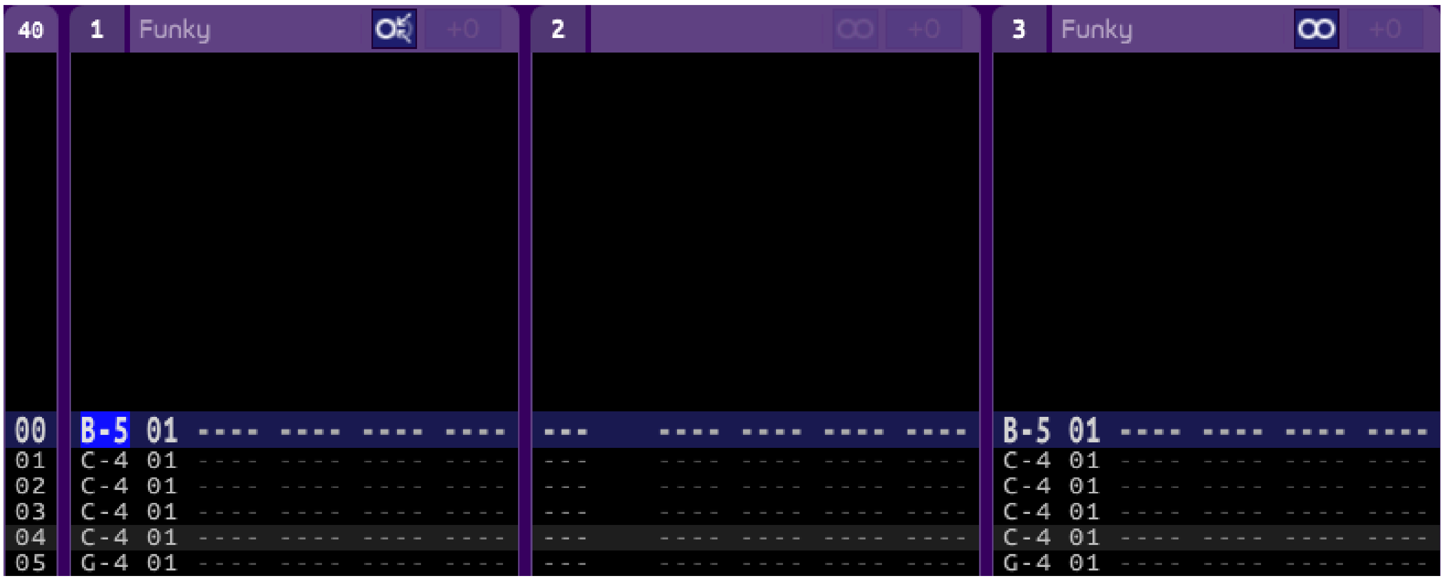

```
40 | 1 Funky          +0 | 2                   +0 | 3 Funky          +0
00 | C-4 01 ---- ---- ---- ---- | ---    ---- ---- ---- ---- | C-4 01 ---- ---- ---- ----
01 | C-4 01 ---- ---- ---- ---- | ---    ---- ---- ---- ---- | C-4 01 ---- ---- ---- ----
02 | C-4 01 ---- ---- ---- ---- | ---    ---- ---- ---- ---- | C-4 01 ---- ---- ---- ----
03 | C-4 01 ---- ---- ---- ---- | ---    ---- ---- ---- ---- | C-4 01 ---- ---- ---- ----
04 | C-4 01 ---- ---- ---- ---- | ---    ---- ---- ---- ---- | C-4 01 ---- ---- ---- ----
05 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ----
06 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ----
07 | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ---- | ---    ---- ---- ---- ----
```

Channel 1 and 3 have the same notes! “Funky” is on both! Notice the icons in the headers:

- On channel 1, the icon means that this track is *linked to*. Some tracks refer to it.
- On channel 3, the icon means that this track is *linked to another one*.

In both cases, this is important, because it means that modifying this track will also have consequence in other parts of the song!

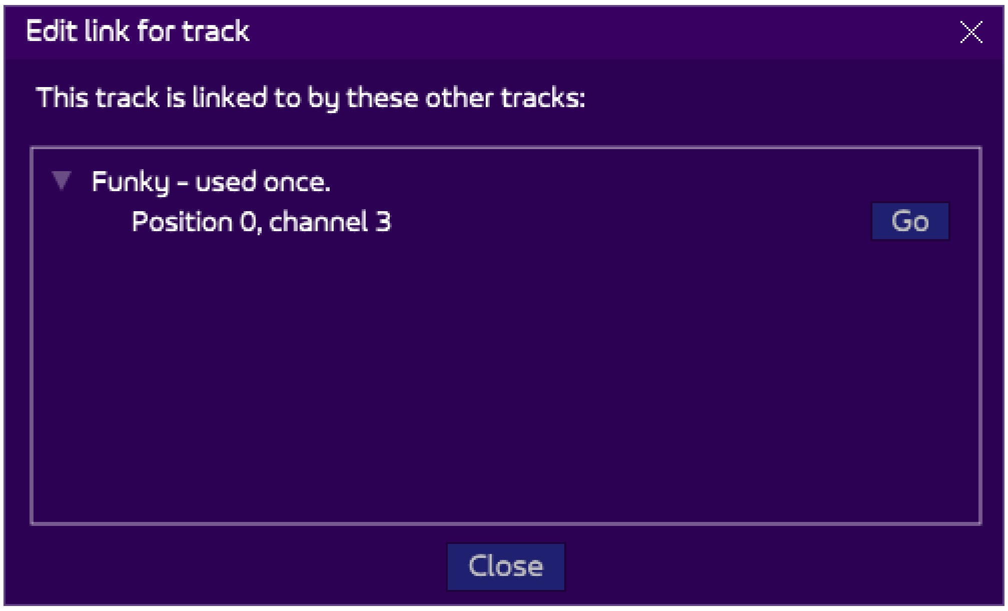
Just try to add one note to channel 1, say, on line 5. And change a note in the channel 3, on line 1:



```
40  1 Funky                  +0   2                          +0   3 Funky                  +0

00  B-5 01 ---- ---- ---- ----  ---     ---- ---- ---- ----  B-5 01 ---- ---- ---- ----
01  C-4 01 ---- ---- ---- ----  ---     ---- ---- ---- ----  C-4 01 ---- ---- ---- ----
02  C-4 01 ---- ---- ---- ----  ---     ---- ---- ---- ----  C-4 01 ---- ---- ---- ----
03  C-4 01 ---- ---- ---- ----  ---     ---- ---- ---- ----  C-4 01 ---- ---- ---- ----
04  C-4 01 ---- ---- ---- ----  ---     ---- ---- ---- ----  C-4 01 ---- ---- ---- ----
05  G-4 01 ---- ---- ---- ----  ---     ---- ---- ---- ----  G-4 01 ---- ---- ---- ----
```

Wow, both changes are applied to both channels! This is the magic of linking.

Let’s click on the icon of the channel 1:



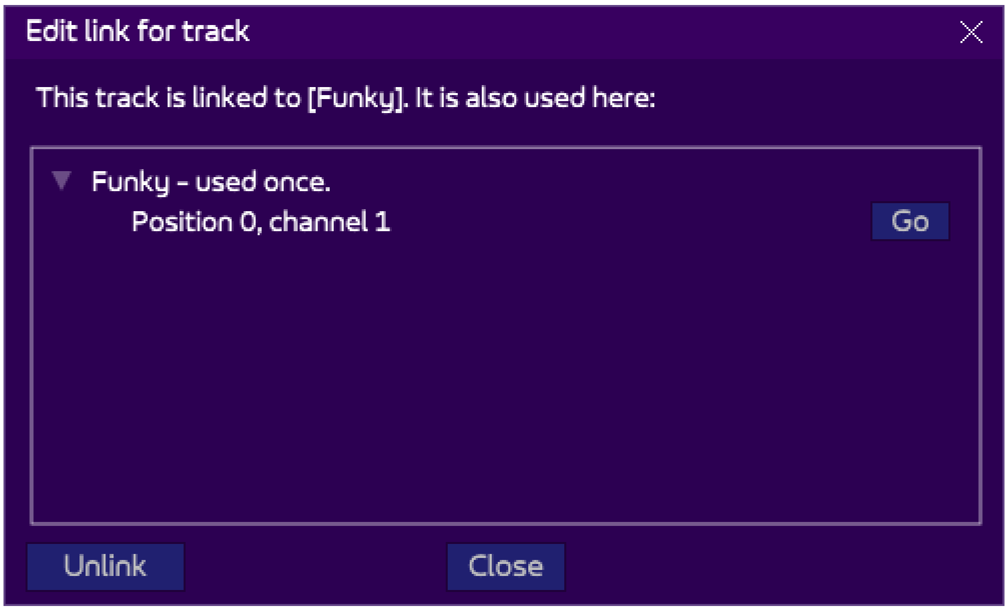
Edit link for track

This track is linked to by these other tracks:

Funky - used once.
Position 0, channel 3

It indicates that the track is *linked to* by one track: the third one in the same position. Note that there is no “link” button here. A *linked to* track can not be linked to another one, that would led to an incomprehensible mess. To do so, you would need to unlink all the referring tracks. We’ll see that in an instant.

Now click on the icon of the channel 3:



This lists the other tracks that are linked to “Funky”, including “Funky” itself. If another track was linked to “Funky”, it would appear in this list too.

Linking is safe

Notice the Unlink button at the bottom-left. If you click on it, the track will return to its original state, that is, the track (and thus the notes!) before we tried to link it to “Funky”.

This is a worthy note: **linking is non-destructive**. You can always undo the linking and “recover” the notes that were there before the linking was done!

Also note that cloning a pattern will, in the new pattern, “convert” the *linked to* tracks into *linked* tracks. This is a desired behavior, as it means you can clone a pattern yet not clone the tracks you decided to share to other patterns, but simply use the link. However, the *linked* tracks remain *linked* tracks.

Linking and transposition

On top of that, you can still use transpositions on *either* or *both* the linked to track and linked track. Transposition is independent to linking, so you can still optimize your song by using linking, yet add variety of sounds by using transposition!

Sound effects

Arkos Tracker 3 has a powerful support for sound effects. They are managed in a different way than Arkos Tracker 1 and 2, to be even more simple to use.

The sound effects **must** be composed in a **separate song**, and each instrument can then be exported as a sound effect, at the desired frequency. This allows to use the same sound effect set among multiple songs (or to be used even without music), thing that was not possible with AT1. The difference with AT2 is a detail explained below.

Preparing the sound effects song

- First of all, create a **new song** or use the default one when launching AT3. Make sure that the only subsong of the song is dedicated to the sound effects.
- Set the right PSG parameters. Especially, makes sure the PSG frequency is the right one. To do that: Edit > Song properties > <your subsong>.

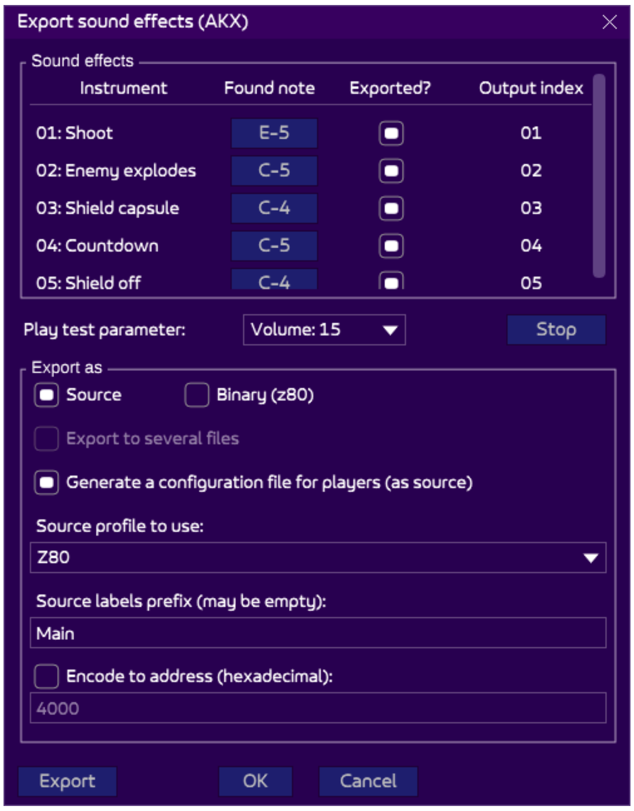
Creating the sound effects

- **Create** as many instruments as there are sound effects.
- **Name them** properly to tell the important sound effects from possible test sound effects. They will not be exported if you don't want to.
- **Use them** in the song: this is needed for two reasons:
    - You can hear what they sound like by simply playing your song (and show them to your fellow programmer conveniently).
    - AT3 will only export instruments that are used (this is a key difference from AT2), and use the note it found first. So you can have fun with a track containing the instrument 01 ("Boom") with the notes `C-2`, `B-4`, `C-5`. But since `C-2` is written first, the exporter will use this note.

Exporting the sound effects.

If you want to test this quickly, you can load test sfxs. You can use either *songs/STarKos/Targhan – Dead On Time – Sound Effects.sks*, or *songs/ArkosTracker2/3Channels/SoundEffects.aks* (both are in the package). We'll use the second in the example below.

To export sound effects to use them in your productions, it's easy, click on File > Export > Export sound effects (AKX). A new window opens:


Export sound effects (AKX)

Sound effects

Instrument	Found note	Exported?	Output index
01: Shoot	E-5	✓	01
02: Enemy explodes	C-5	✓	02
03: Shield capsule	C-4	✓	03
04: Countdown	C-5	✓	04
05: Shield off	C-4	✓	05

Play test parameter: Volume: 15

Export as
[✓ Source] Binary (z80)
Export to several files
[✓ Generate a configuration file for players (as source)]
Source profile to use: Z80
Source labels prefix (may be empty): Main
Encode to address (hexadecimal): 4000

The list at the top shows all the instruments of the song. They are all considered "used" if AT can find at least one note in your sfx song, or marked "Unused" otherwise. However, you can remove any sfx you don't want by unticking the "*Exported?*" checkbox.

You can see that a sound effect relates to an "export note": this is the note at which the sound effect will be played, and it directly extracted from your sfx song. As explained before, the first note found is what matters.

By clicking on the note of each sound effect, it is played in the note AT has found. You can change the volume it is played at in the "*play test parameter*" below. Note that it is only a test parameter, it does not change how the sounds are exported.

The "*output index*" besides each sound effect indicates the index to use when playing the sound effect. Indeed, unused and unexported instruments may create holes in the list, but they are actually skipped, making the sound effect output indexes linear.

The "*generate a configuration file for players*" should be checked: on export, a second tiny file will be generated, indicating to the player what features your sound need. The ones you don't won't be compiled, saving both CPU and memory! More information here.

When you are satisfied with the sounds effects, you can press "*OK*" to save these new parameters and exit, or "*Export*" to export them for the sound effect player to use.

Using the sound effects

After you have clicked on "Export", generating either a binary or an AKX source file, it is time to play them on hardware.

Technical note: the sound effect player is somewhat decoupled from the players. Ideally, I wanted to have one sound effect player for all the available players. But this has proved inefficient: each player has its specificity. So each player has its sound-player counterpart, in another source file.

The sound effect player is always linked to a player: it may be AKY, AKG, AKM or Stand-alone. Finds the one you need and activate the sound effects in the assembler code, at the beginning of the source. This is usually done by simply adding a flag in your source, such as:

```
PLY_AKG_MANAGE_SOUND_EFFECTS = 1
```

The label is "AKG" here, but of course, this changes according to the player (AKY, AKM, SE for Sound Effects).

All is explained in the player source, don't worry!

Initialization

Just like the music player, the sound effect player must be initialized. It's easy:

```
ld hl,soundEffects
call PLY_AKG_InitSoundEffects
```

The "soundEffect" points on the data of the exported sound effects, which is either pure binary or source file.

Initializing the sound effect player can be done at any time, as long as it is done **before** playing a sound effect: you can even do it before initializing the music player.

Playing a sound effect

You have your music running and your sound effect player initialized. The character of your game shoots: let's call the "shoot" noise!

```
ld a,soundEffectNumber ;(>=1)
ld c,channel ;(0-2)
ld b,invertedVolume ;(0-16 (0=full volume))
call PLY_AKG_PlaySoundEffect
```

That's it! The "soundEffectNumber" is the number, as seen in the Export window above: it is simply the "output index" of the instrument to play. For example, to play "Enemy explodes", set A to 2.

The "channel" is from 0 to 2 and defines on which channel the sound is played.

The "invertedVolume" is the volume of the sound effect, from 0 to 16. 0 is the loudest, 16 is mute. Why 16 do you ask, as the PSG only has a volume range from 0 to 15? Because hardware sounds are also faded by this volume: they are considered a 16th volume. But fading them will also make them lose their "hardware" status: they *will* sound differently. But you will be able to fade them, at least.

One important thing to understand is that the *PLY_xxx_PlaySoundEffect* method only "triggers", or "programs" the sound effect. In itself, it does not do anything more than telling the player "this sound must be played". But the sound effect will only be played, frame by frame, each time the *PLY_xxx_Play* method of the player is called.

Stopping a sound effect

Your sound may loop and you want to stop it, or the player has pressed "esc" and you want to mute all the channels. This is possible:

```
ld a,channel ;(0-2)
call PLY_AKG_StopSoundEffectFromChannel
```

If you want to stop all the channels, call this method three time using the 0, 1, 2 values.

Sound priority

If a sound is played on the channel 1, playing another sound on the same channel will take priority on the previous sound.

You may want to manage priorities on the sound effects: for example, an explosion sound can not be stopped by a bullet firing sound. It’s is not managed by the player, so you have to do it by yourself.

Sound effects without music

You may want to play sound effects but don’t need music in your production. There are two ways of doing it:

- Use the stand-alone sound effects player, in the AT3 kit.
- Generates an empty music and use one of the player to play it (of course, nothing will be heard). Use the related sound effect player to play your sounds. There is of course big overhead since you integrate a full music player that is not required! However, this is a convenient solution if you already have a player at hand (AKG, AKM, AKY), no need to integrate a second one.

The streamed music analyzer

Did you ever wonder how such artists created this or this sound? How could an AY or a YM produce such a beautiful noise from 3 simple channels? Well, the answer is now within reach.

The streamed music analyzer (SMA) is a tool which loads streamed-music such as YM (a format created on ST by Leonard/Oxygen) or VGM. These consists of a stream of PSG frames, which make them very easy to read and play (but their memory footprint makes them all the more difficult to use on the real hardware without resorting to clever compression).

Once loaded, the SMA allows you to play the song, and even better, extract part of it into Arkos Tracker instruments! You will now know the secrets of all these musicians you've admired!

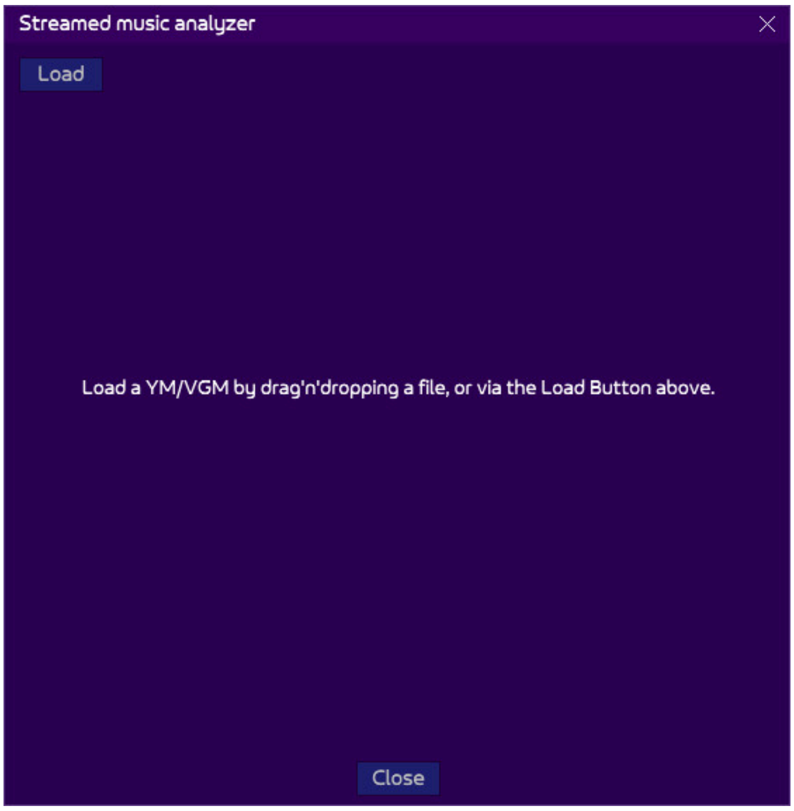
Getting the music

First of all, you must get hold on YMs or VGMs. The latter is a widely known format in the arcade community, so it's very easy to get a such files, though actually less on the PSGs we're using.

As for YM, you can get a whole archive here (Atari ST). You can also use an emulator (such as ACE-DL or Winape) to record the music of a production and load it back into AT.

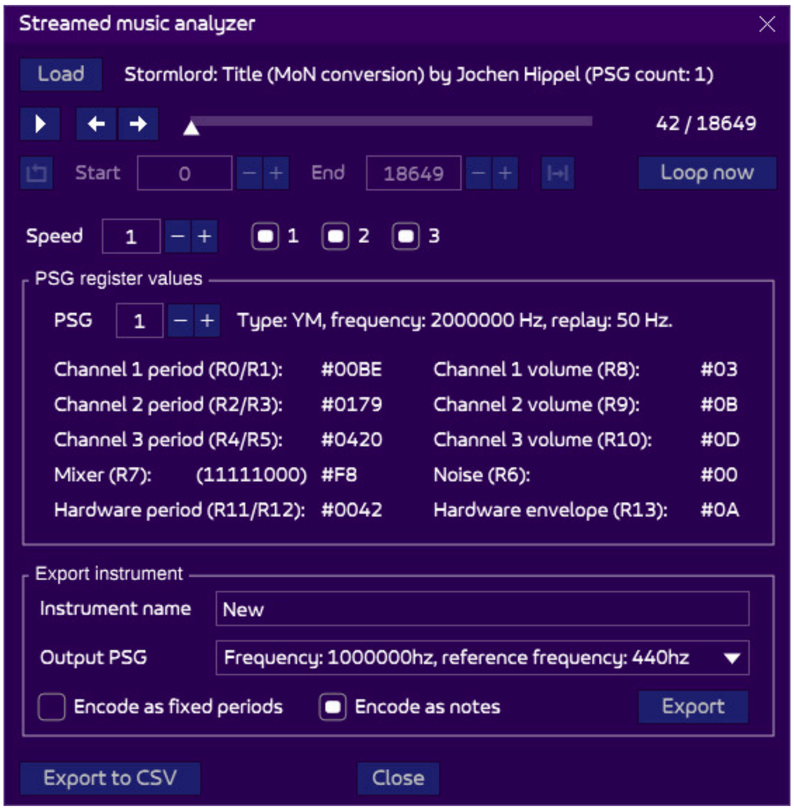
Loading a song

Once you have your music, open AT and the Tool > Streamed music analyzer.



Load a YM/VGM by drag'n'dropping a file, or via the Load Button above.

As the text indicates, you can either drag'n'drop the file onto the windows, or open use the Load button. Once loaded, the window will look like this:



Stormlord: Title (MoN conversion) by Jochen Hippel (PSG count: 1)

42 / 18649

Start 0 End 18649

Speed 1 ☑ 1 ☑ 2 ☑ 3

PSG register values

PSG 1 Type: YM, frequency: 2000000 Hz, replay: 50 Hz.

Channel 1 period (R0/R1):	#00BE	Channel 1 volume (R8):	#03
Channel 2 period (R2/R3):	#0179	Channel 2 volume (R9):	#0B
Channel 3 period (R4/R5):	#0420	Channel 3 volume (R10):	#0D
Mixer (R7): (11111000)	#F8	Noise (R6):	#00
Hardware period (R11/R12):	#0042	Hardware envelope (R13):	#0A

Export instrument

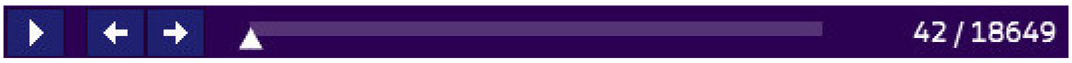
Instrument name: New

Output PSG: Frequency: 1000000hz, reference frequency: 440hz

☐ Encode as fixed periods ☑ Encode as notes

There are several parts, explained right below.

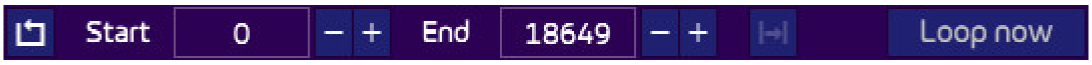
The play part



42 / 18649

The Play button is pretty self-explanatory. It will play the whole song, or the loop if it is activated (see below about it). The two arrows will move to the previous or next frame. You can move the cursor with the mouse to reach specific parts of the song, but the range is limited by the loop, if activated. On the far-right is the number of the currently played frame, and how many there are in the song.

The loop part

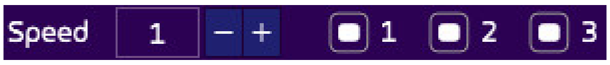


The loop is crucial to listening to the exact part you want to analyze or extract. If disabled, you are free to explore the song within its limit. However, once the loop is enabled (far-left button), only the loop is played.

The “Loop now” button is very handy: it sets the start/end of the loop around the currently played frame. My advice on how to use it is play the whole song (i.e. without the loop on), and whenever you hear a sound you want to analyze/extract, press “Loop now”. It automatically turns the loop on. This is particularly efficient to capture a drum or a short sound.

The icon on the right, dubbed “locked length”, will make sure, once enabled, that changing the start also shifts the end accordingly.

The speed/channels part

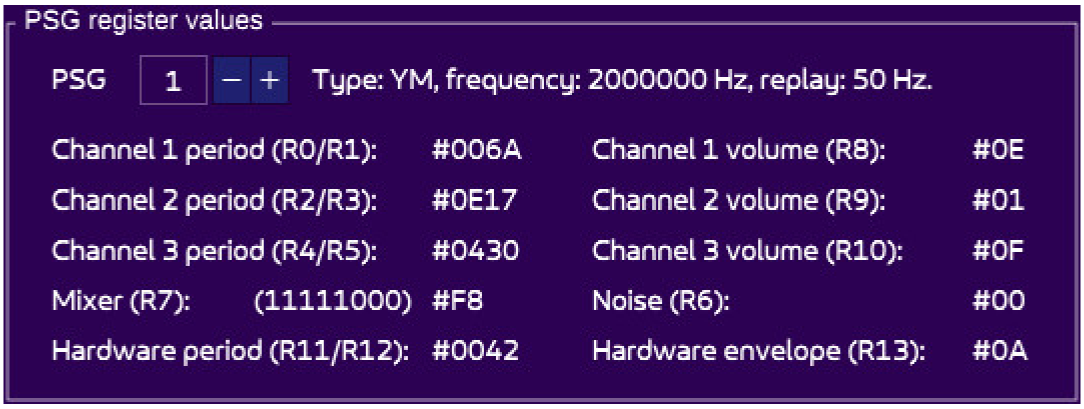


A speed of 1 is the original speed. This is nice to listen to the song or reach the part you’re interested in, but often you will want to slow things down to understand what is going on. Using a higher number will slow the replay down. **This has no effect on the export.**

The numbered toggle-boxes on the right mutes/un-mutes the channels. This has two uses:

- Listen to a particular channel to better understand its secrets.
- An export is done only from one channel only, that is, the one remaining channel that is on.

The registers part


PSG register values

PSG 1 Type: YM, frequency: 2000000 Hz, replay: 50 Hz.

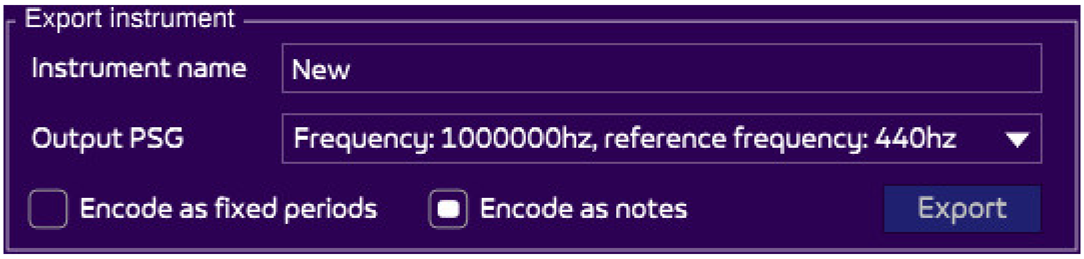
Channel 1 period (R0/R1):	#006A	Channel 1 volume (R8):	#0E
Channel 2 period (R2/R3):	#0E17	Channel 2 volume (R9):	#01
Channel 3 period (R4/R5):	#0430	Channel 3 volume (R10):	#0F
Mixer (R7): (11111000)	#F8	Noise (R6):	#00
Hardware period (R11/R12):	#0042	Hardware envelope (R13):	#0A

This is a highly technical part, the one that may hurt your eyes at first! It shows in real-time the currently played PSG frame, encoded in the loaded format. Note that this is a raw dump: many YM/VGMs contain useless bits, probably because the player on the hardware was strangely coded!

Use the PSG slider to select a PSG, but know that YM music have **only one**, the VGM **only 2** (for AY/YM. It may contain more with other PSGs, but they will be discarded).

Understanding these raw values require a documentation of the AY/YM, which is easy to find in this day and age, but not mandatory to extract the sound that picked your interest.

The export part


Export instrument

Instrument name: New

Output PSG: Frequency: 1000000hz, reference frequency: 440hz

Encode as fixed periods [✓ Encode as notes]

You have targeted a range you want to export, to use it in your song or analyze it further in the instrument editor.

Before pressing Export, enter the name of your new instrument. Then, check the output PSG. It is automatically set on the PSG that was pointed by the cursor in the Pattern Viewer. Most of the times, it is a single PSG, but if you're working on a multi-PSG song, it has its importance, especially on such hardware as PlayCity, as the two additional PSGs can have a frequency up to 2Mhz, so different from the original one. Playing a sound on a 1 Mhz PSG *may* sound different, all the more when exporting as fixed periods, as explained just below.

There are two types of export:

- Encode as fixed periods: the generated instruments will use hardcoded periods: it means that you will always hear the same sound regardless of the note you enter. This is handy for drums, as you can trigger them, and use the transposition of the track. It will be applied to all the instruments, except these hardcoded-period sounds! Warning, there is a drawback for such sounds: they *will* sound different from a PSG to another if their frequency is different! Hence the use of the Output PSG setting. A period of 239 is a C-4 on a 1 Mhz PSG, but a C-5 on a 2 Mhz PSG!
- Encode as notes: the periods are all using the “automatic” setting of AT. If a period is not an “official one” (say, a C-4 but slightly higher or lower), a pitch is also added. This setting is relevant for melodic sounds. It can even convert arpeggios!

Last options

The “Export to CSV” button is useful to examine the data with your favorite CSV reader, such as Excel or VSCode. It is also useful to export them back into another format of your own.

One last option worthy of note is that if you close the window, opening it back will load the song where you were, with the loop parameters loaded. This is convenient after you’ve exported an instrument, want to tweak it, then go back to plundering the same YM.

About imports

A lot of effort has been put into the import of various file formats. However, please bear in mind that imports are **best-effort**. Except for STarKos (SKS) and Arkos Tracker 1/2 which formats are quite analogue to the one of AT3, each music software has its capabilities and design, which make it sometimes very difficult to translate into another tracker.

Import is not made to get any of your <*enter your decrepit tracker here*> song and use the AT3 players: you will probably have to make some adjustments. If some features are really missing, please contact me (by mail or via the forum) and I will see what I can do. But import is mostly useful for musicians wanting to finish their song with AT3, and make a definite switch to AT3. Please don't look back to your previous software :).

Import currently supported:

- AKS (Arkos Tracker 1, 2 and 3 (obviously))!
- SKS (STarKos)
- 128 (BSC's Soundtrakker 128)
- MOD
- MIDI
- WYZ Tracker
- CHP (Chip'n'sfx)
- VT2/TXT (Vortex Tracker 2 – PT3 can be converted by VT2.5)

MIDI import

The midi import allows to… import MIDI files.

Note: Steph' has written an article in French about this (with AT2, but it works with AT3 the same way), with practical examples. Thanks a lot to him!

Features

- Program Changes are converted into basic AY instruments.
- Drums are also converted.
- Note velocities can be converted.
- BPM changes are converted.
- Time signature are used to determine the height of the patterns.

All the tested files worked quite well, but there are some constraints inherent to this format: MIDI is not always structured into tracks, and is definitely not structured in patterns, as trackers are. AT2 tries, thanks to the time signature, to split the Midi events into patterns, and does a good job at it most of the time.

Recommendations

- Use MIDI 1 format (multi-track), not MIDI 0, as it has only one track.
- Have **one instrument per track only**.
- Have your notes well locked to bars and beats (avoid humanization). Warning, some MIDI files have their notes completely out of sync with their BPM. It will NOT work well for the import.
- Time signature helps (4/4 is default), because it helps define the height of the patterns.

Parts per quarter (PPQ)

When importing, the software will ask for the PPQ (parts per quarter) to use. It defines how accurate the importing is done. Each MIDI song has its own PPQ (960 most of time). In order to have one "Tracker line" per quarter-beat (and thus, 4 lines per beat), the default import PPQ will be the song PPQ / 4 (so, 240 if the song PPQ is 960). This may be enough for most songs. If not, use a smaller value, but it should be a multiple of the song PPQ for a better result (240, 120, 60, 30, etc.).

As a result, the lower the chosen PPQ is, the more "lines" will be generated. The speed is also doubled accordingly, but remember that the speed is not as accurate as BPM!

Like any import, MIDI songs will have to be tweaked, but a good import should generate all the right notes, so tweak the PPQ if the result is not to your liking.

MOD import

The MOD import works quite well and can import both 15 and 31 instrument MODs.

Apart from that, it is quite limited, effect-wise: only Volume is imported. Since no one asked for more, it won’t evolve, unless someone requests it.

VT2 import

The Vortex import works on **.txt** (generated by Vortex Tracker 1) or **.VT2** (generated by Vortex Tracker 2.5), which is in fact the same format with a different extension.

PT3 files can not be imported directly, but by saving them as .txt (in VT1) or as VT2 (in VT**2.5**), this works all the same.

The import is raw but works quite well. However, due to some huge difference in how the patterns and instruments are designed, some parts were discarded on purpose:

- The hardware sound are **not** managed. This is because all the hardware management is done on the instrument level in AT3. I could make a more advanced importer, but in many case, the result would not be satisfactory. So I preferred removed this feature, and the musician can modify the imported song to make it sound right.
- The noise and envelope columns in the patterns are not managed at all, for the same reason.
- The pitch, volume and noise increasing/decreasing in the instruments are managed, by generating lines (up to 64). This is not the natural way AT3 works, but at least simple sounds (such as drums or decaying sounds) are converted correctly. Once again, it is up to the musician to adjust the sounds afterwards if he’s not satisfied.

Export to MOD

The Export to MOD feature records your sounds as WAV and produces the patterns so that you can continue your song with any MOD editor, such as OpenMPT.

However, please bear in mind the limitations of the exporter, but especially the MOD format!

Limitations of the exporter

- Only the Set Volume and Reset effects are encoded.
- … that's all :). No so bad, he ?

Limitations of the MOD format

- Only 31 instruments!
- Only 64 patterns!
- Only 5 octaves! Out-of-bounds octaves are capped.
- The samples must be 8373 Hz, so the PSG sounds will probably sound bad.
- The samples are limited to 64k.

So you might want to use MOD as a pivot format, and rework the whole thing. You could also export the instruments you really want, one by one, using the Instrument To WAV export, at a high frequency (such as 44100 Hz), and replace the ones from the export.

How samples are managed

Overview

Arkos Tracker can handle samples natively in two ways:

- via events.
- via instruments, directly in notes in the pattern.

The events are used to play single notes, non-looping samples, which make them ideal for drums, also called "digidrums". The AKY player has a special sub-player to play them.

The sample instruments, written as notes in the pattern, are more complex to handle, depending on the context:

- For sample-only music, a MOD player is available.
- For PSG+sample, using a specific player.

Samples, on most 8-bit machine, have an important cost:

- When triggered, they use 100% of the CPU. Most of this time is actually spent in waiting (except for MOD where every cycle is used!), so if you want to integrate an effect, it is up to you to "micro-code" your effect within this tempo. It is a possibly very difficult task, so please bear that in mind before starting asking your coder about using digidrums. Checking for the keyboard is ok though.
- Depending on your assets, each sample may take at least 1kb for drums, any a few kilobytes for longer sounds.
- Making the samples sound ok along the song may need a bit of tweaking of volumes, frequency filters, so be patient.

Sample frequency and sample player frequency

This part is important if you want to take the full advantage of AT sample capabilities, and understand how to make your samples sound right when targeting the hardware.

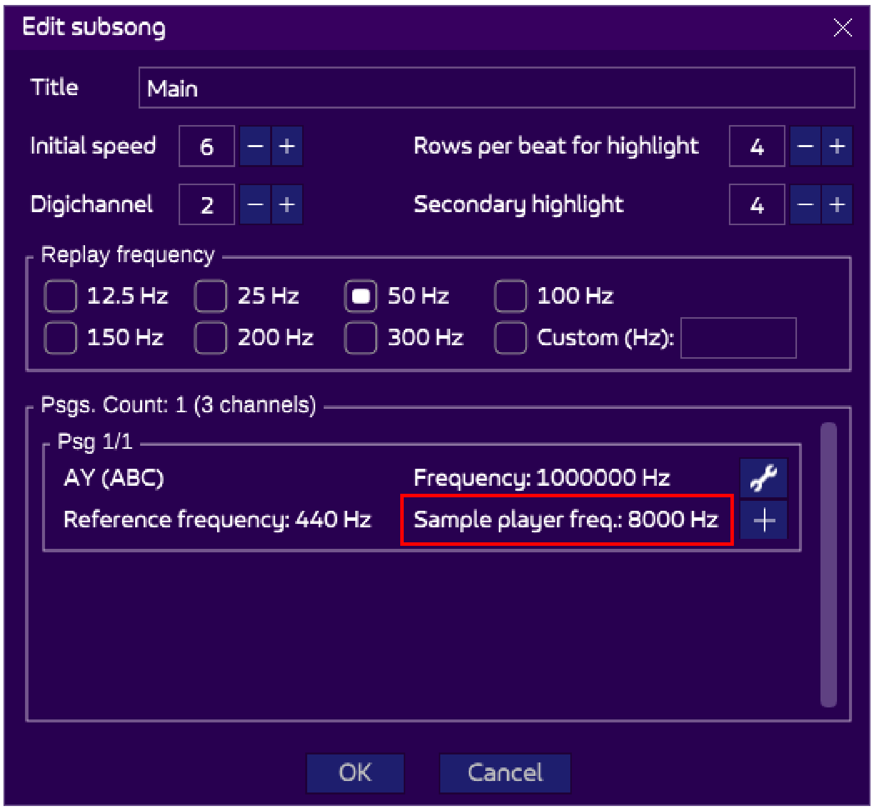
Sample frequency

First of all, your sample has been sampled using a frequency. On a modern computer, you would most likely find samples sampled at 44100 Hz, or sometimes even more. On old MODs, you would be likely to find samples at 8373 Hz. Without going into details, this means that such sample will lose all their high-end frequencies, but this will also save a lot of memory.

So if you need to include samples into your song, do NOT use 44100 Hz, unless you know what you are doing, because the sounds will fill your memory very quickly! Depending on the usage, I would suggest using 8000 Hz samples (and *not* 8373 Hz) if your hardware is a 8-bit and for drums. You may want to increase the quality to 11025 Hz samples if you play a MOD.

Sample player frequency

This concept is new in AT3 (it was present in AT2, but not handled the same/proper way). In the PSG settings of your subsong, a "sample player frequency" is shown, with a default to 8000 Hz:


Edit subsong

Title: Main

Initial speed: 6

Rows per beat for highlight: 4

Digichannel: 2

Secondary highlight: 4

Replay frequency: 12.5 Hz [✓ 50 Hz] 25 Hz 100 Hz 150 Hz 200 Hz 300 Hz Custom (Hz):

Psgs. Count: 1 (3 channels)

Psg 1/1

AY (ABC)

Frequency: 1000000 Hz

Reference frequency: 440 Hz

Sample player freq.: 8000 Hz

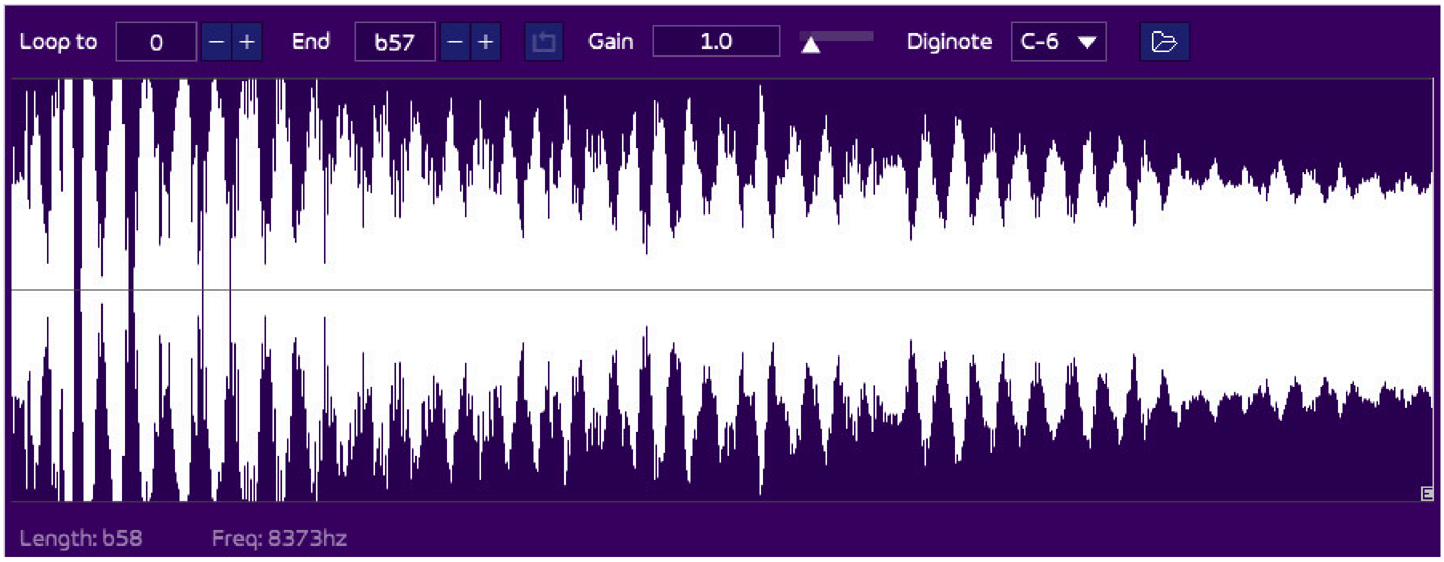
This is an important concept, because it says to AT3 that the "code" that plays a sample has a frequency of 8000 Hz. So it will play a 8000 Hz sample perfectly using the "reference note" (which is C-6). If your sample is sampled at 16000 Hz, you will have to play it twice faster (C-7) to play it to its natural note.

On the opposite, if your sample player frequency is high (because you are on a Atari ST maybe, or have a kick-ass replay routine), like 16 kHz, you will probably have to use lower notes to play 8000 Hz samples right.

The bottom line is this: when wanting to play samples, ask the coder what is the sample player frequency. It will probably be 8 kHz on an 8-bit. By knowing that, you will know what note to use when composing the song. Don't worry: if you or your coder make a mistake, it is still possible to correct the sounds by transposing them in the patterns, or upload new samples.

Diginote

The diginote, in the sample editor, has a default note of C-6.


Loop to 0 End b57 Gain 1.0 Diginote C-6

Length: b58 Freq: 8373hz

This indicates the note used when a sample is played in the **event track**. However, it has a hidden consequence: when exporting the samples, the sample is **resampled** according to this note, so that it sounds right on the hardware. **Warning**, the sample is resampled even if used in a pattern, so it is advised not to change the diginote with samples used in patterns (anyway, the diginote has no consequence to them, so you'd have no interest in changing it).

Sample export

The sample export is possible for formats that allow samples, such as AKG or RAW.

The options are numerous to fit all the needs, as they may be very different from a hardware to another.

- Amplitude (8 bits): this sets the maximum amplitude in the generated samples, from 1 to 256. If you want 8 bits samples, enter 256. For 4 bits samples, enter 16. Basically, this is the value that will never be reached for the sample. The one below will (256 : values from 0 to 255, 16 : values from 0 to 15). More esoteric amplitudes are accepted. For example, my sample code on CPC requires an amplitude of 10 (values from 0 to 9).
- Offset (8 bits): this adds an offset to the generated values of the samples, **regardless of the bit rate**. For example, if you add an offset of 128 to an amplitude of 16, values will go from 128 to 143, included. This may be once again useful for specific trick depending on the hardware (on CPC, we can optimize the PSG code by having an offset of 128).
- Padding length: adds a padding after the data of the sample. Its length is in bytes. This is useful to make sure no “rubbish” data will be heard if your replay code do not detect with byte-accuracy the end of a sample. Note two things:
  - For **non-looping samples**, the padding byte is the “middle” value, according to your amplitude, by default. You may want it to be 0 with the option “padding/fade to min value” below.
  - For **looping samples**, the padding is actually a copy of the looping part of the sample, so that your replay code can loop seamlessly even if going “beyond” the sound, as the padding will be there to play the data it is supposed to play.
- Fade-out length: Only used for **non-looping samples**. Applies a fade-out on the last bytes of the samples. How many is defined by the “length”. 0 means no fade out is applied. This is especially useful to avoid “clicks” when a sample is over (especially with digidrums). A value of 20 will probably be enough. Note that the padding, if present, is not influenced by this option. The target byte of the fade-out is the “middle” sample value, but can be 0 by setting the option “padding/fade to min value” below.
- Padding/fade to min value: If set, the padding value is 0 as well as the fade-out target value. If not set, the value is the “middle” one.
- Export only used length: **Ignored for looping sounds with padding option on.** If not set, the full length of the sample is exported, even if only part of it is used (via the “end index” of a sample). If set, only the data from the beginning to the “end” value is exported. This is useful if you

imported a sample, but then realized it could be shorter, so in the sample viewer, you moved the “end” marker to the left.

Digidrum player

AT3 allows you to play digidrums, that is, usually small samples, tune-less (it is not possible to change the note). The only player that can play them is the multi-PSG variant of the AKY player (in the package alongside the AKY player).

Digidrums are available for Amstrad CPC, MSX and Spectrum out of the box (it you need more platforms, contact me. It also should work on the first PSG of multi-PSG hardware). Testers are available for you to try out (*PlayerAkyDigidrumsTester_CPC.asm*, for example).

The digidrums are triggered via the **event tracks**. They are **NOT** triggered from sample notes (there is another player for that). Also, please read about the cost of samples here before integrating blindly to your song :).

Playing digidrums is a 3-step process, provided that you have a song with digidrums triggered via the event tracks:

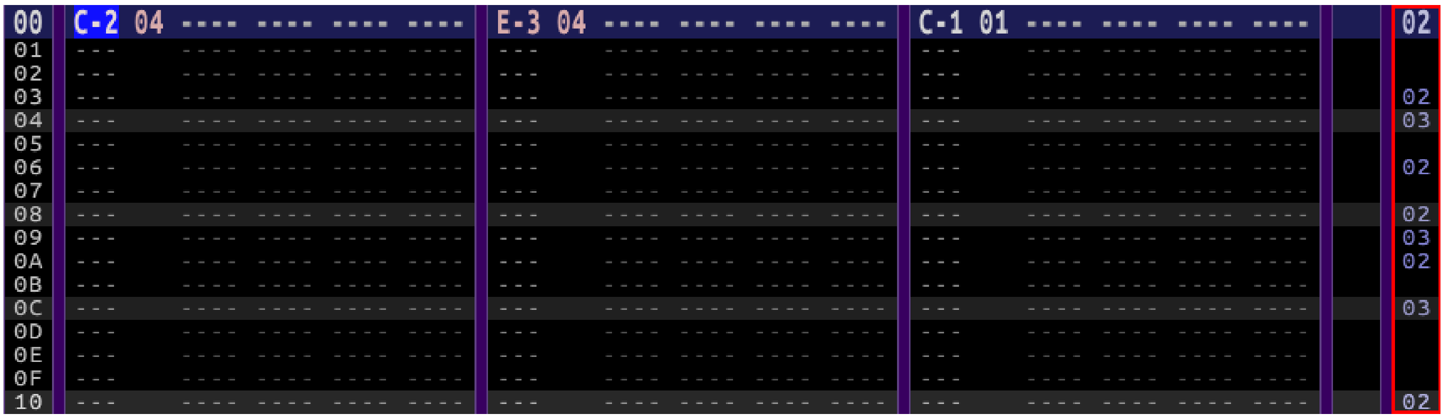
- Export the song to AKY.
- Export the events themselves, exported via Events Export.
- Export the samples, exported via Sample Export.

Do not worry, each of these steps are simple exports from AT, with just simple checks, and are explained below. You can also make the export via the command-line tools, automatising the process once and for all (more on this below).

Export via the software

A simple example is already provided with AT3: it is the *Digitest* song in *songs/ArkosTracker3/digidrums/Digitest* (you can also use the *Sarkboteur* song in the ArkosTracker2 folder). It is a simple song with two digidrums (Bassdrum (02) and Snare (03)).

Both samples are 8 kHz, and triggered via the event tracks:

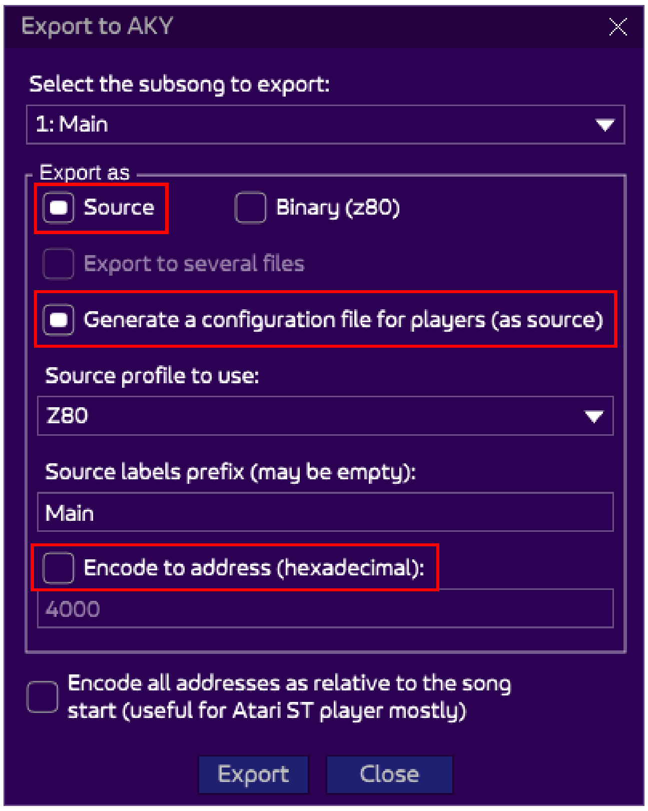


```
00  C-2 04 ---- ---- ---- ----  E-3 04 ---- ---- ---- ----  C-1 01 ---- ---- ---- ----  02
01  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
02  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
03  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  02
04  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  03
05  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
06  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  02
07  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
08  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  02
09  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  03
0A  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  02
0B  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
0C  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  03
0D  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
0E  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
0F  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----
10  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  ---    ---- ---- ---- ----  02
```

Following the three steps mentioned above:

Export to AKY

This is simple and there is nothing to configure, except export as source **without** encoding an address. Also tick the "generate a configuration file", though this is optional:



Export to AKY

Select the subsong to export: 1: Main

Export as: [✓ Source] Binary (z80)

Export to several files

[✓ Generate a configuration file for players (as source)]

Source profile to use: Z80

Source labels prefix (may be empty): Main

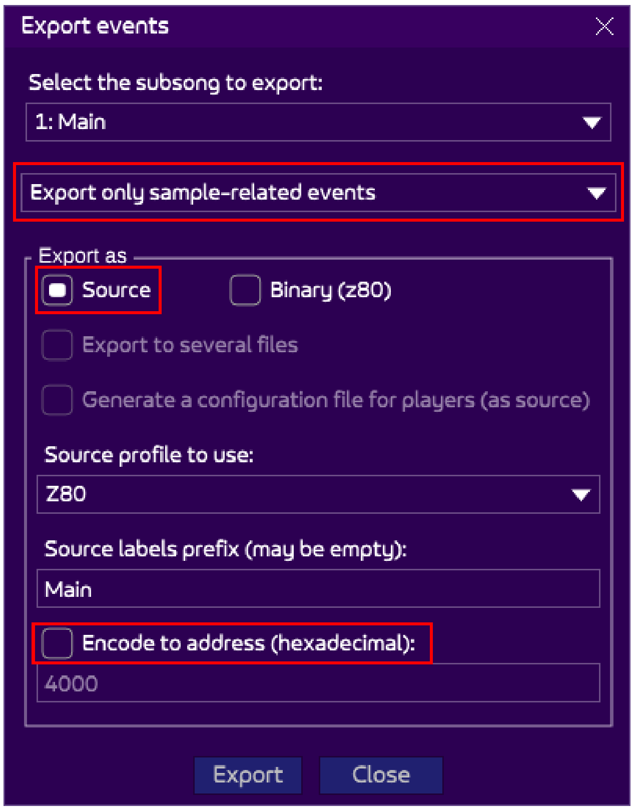
Encode to address (hexadecimal): 4000

Encode all addresses as relative to the song start (useful for Atari ST player mostly)

Export   Close

Export events

Use the "export events" option, with these values:


Export events

Select the subsong to export: 1: Main

Export only sample-related events

Export as

[✓ Source] Binary (z80)

Export to several files

Generate a configuration file for players (as source)

Source profile to use: Z80

Source labels prefix (may be empty): Main

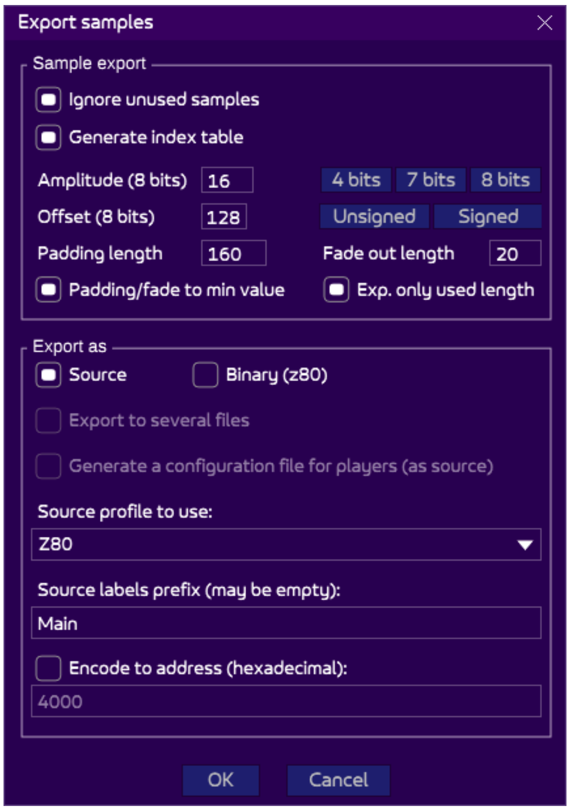
Encode to address (hexadecimal): 4000

Export Close

The “export only sample-related events” is important because events can also be used to synchronize events to your productions. You don’t want these in your digidrum player! By keeping only the “sample-related events”, only events which has the same number as a sample will be taken in account.

Export samples

Finally, export the samples:


Export samples

Sample export

- [✓] Ignore unused samples
- [✓] Generate index table

Amplitude (8 bits): 16 — 4 bits | 7 bits | 8 bits

Offset (8 bits): 128 — Unsigned | Signed

Padding length: 160 — Fade out length: 20

- [✓] Padding/fade to min value
- [✓] Exp. only used length

Export as

- [✓] Source
- [ ] Binary (z80)
- [ ] Export to several files
- [ ] Generate a configuration file for players (as source)

Source profile to use: Z80

Source labels prefix (may be empty): Main

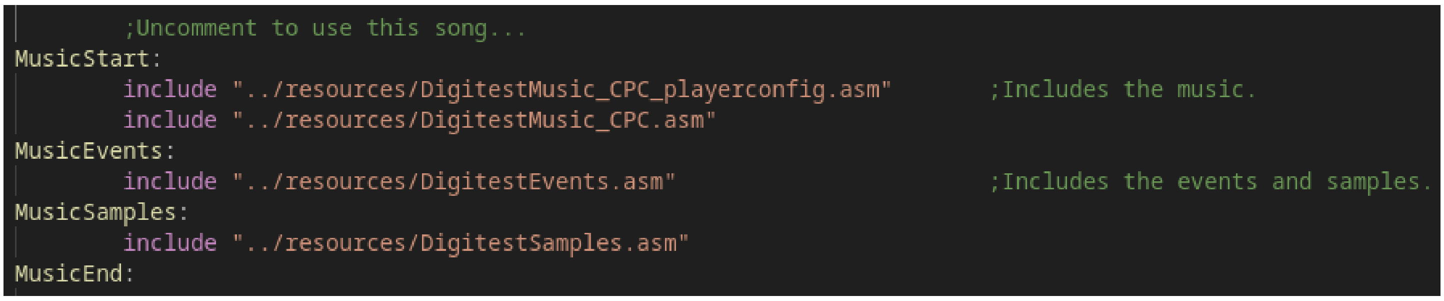
- [ ] Encode to address (hexadecimal): 4000

Besides the usual "export as" at the bottom, the top section is important. You must follow the values stated in this screenshot:

- Ignored unused samples: ticked, to remove the useless data (why waste memory?).
- Generate index table: so that the player knows all the sample addresses.
- Amplitude: 16, because the samples are on all 4 bits (0-15), maximizing their quality.
- Offset: 128, the player relies on an optimization when addressing the PSG. By setting the bit 7 to 1, we save a few cycles.
- Padding length: as explained in the digidrum source, extra-bytes must be added, because the player does not stop the sound accurately (checking it would be too slow), so we add a padding at the end to make sure that no "crappy" values are sent to the PSG. The value 160 depends on how fast the player reads the bytes.
- Fade-out length: 20, to prevent "clicks" at the end of sample.
- Padding/fade to min value: ticked, so that the "zero" is the same zero as the PSG, once again to prevent clicks and various volume problems.
- Export only used length: ticked, as an optimization.

Listening to the result

You should have 4 files (or 3 if you didn't export the "player configuration" in the AKY export). Feed them to the digidrum tester, replacing the currently used files:



```
        ;Uncomment to use this song...
MusicStart:
        include "../resources/DigitestMusic_CPC_playerconfig.asm"       ;Includes the music.
        include "../resources/DigitestMusic_CPC.asm"
MusicEvents:
        include "../resources/DigitestEvents.asm"                       ;Includes the events and samples.
MusicSamples:
        include "../resources/DigitestSamples.asm"
MusicEnd:
```

Compile it with Rasm, and enjoy the result!

Export via command-line tools

More streamlined than clicking on various buttons in AT3, the CLI tools are powerful and allow you to create a build chain, reusable at will. They are in the "tools" folder.

For explanations about the parameters, check the sections above.

Export to AKY

```
SongToAky --exportPlayerConfig <input song> <output AKY file>
```

Export events

```
SongToEvents --eventType sample <input song> <output events file>
```

Export samples

```
SongToSamples -sma 16 -smo 128 -smfo 20 -smpl 160 -smmin -smonly <input song> <output samples file>
```

Listening to the result

Please refer to the section above :).

Sample+PSG player

A player with the possibility of playing **one** channel (at the time) of sample (with pitch variation) along with a 3-channel PSG music is being developed right now. It should be released in the next version (3.2.8).

Wait and see!

Music on hardware in real-time

Wouldn’t it be great if you could hear your music on the real hardware, in real-time?

The answer is yes, and here is a video to prove AT can do it *(note: this video was made for AT2, for AT3 works just the same)*:

Arkos Tracker 2 - Real-time communication with a retro-computer
https://youtube.com/watch?v=wZ8rREBnknM

You can use AT3 to compose fluently on your PC/Mac, but hear the result on your Amstrad/Msx/Spectrum/etc., in real-time. Because as good as the PSG emulation is, your music may still sound a bit different on the hardware. The speaker, the bulk of the machine itself have consequence on how the sound is played on the hardware, which is almost impossible to emulate.

So it is possible to do such communication, thanks to any **serial interface** that your hardware is fitted with. Most old-school computers don't have one by default, but many are now available via an extension port. Of course, it works regardless of your OS (Windows, Linux, Mac).

AT3 itself is compatible with any **serial interface**, the only thing that needs to be done is adapt the Z80 client on the old-computer-side. As of now, **the Z80 client only supports Amstrad CPC interfaces** because I don't have any others! Please contact me if you want a specific hardware to be used, it is easy to do! You can also do it yourself, the Z80 sources are easy to adapt.

The Z80 client currently supports the following serial interfaces:

- Albireo
- USIfAC
- CPC Booster
- Mini Booster

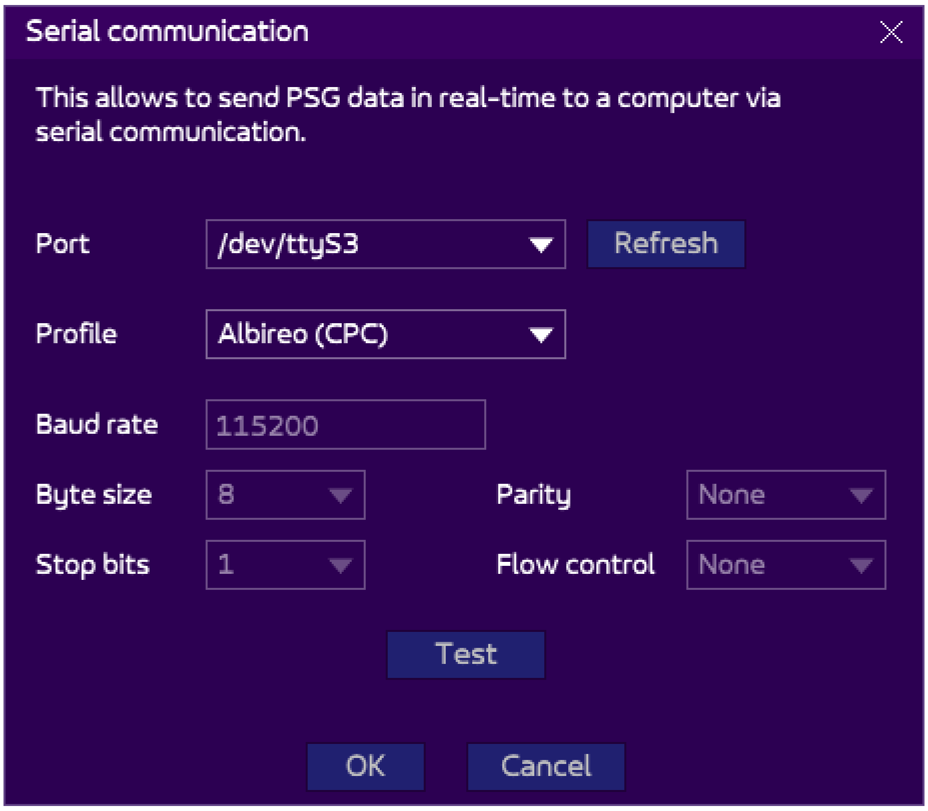
It supports the following PSG:

- Normal PSG
- PlayCity (9 channels!)

Setup

Start AT3, turn your old computer on, with its serial interface. Link both computers with a serial cable according to your hardware, such as a USB cable.

We must tell AT3 on what port to send the music data. Go to File > Setup > Serial communication:


Serial communication

This allows to send PSG data in real-time to a computer via serial communication.

Port: /dev/ttyS3
Profile: Albireo (CPC)
Baud rate: 115200
Byte size: 8
Parity: None
Stop bits: 1
Flow control: None

First, select a Port. On Linux, it may be something like "dev/…", on Windows "COMxx".

Once this is done, select a hardware profile. Some interfaces are already present (Boosters, Albireo, USIfAC…), but you can also use a "custom" profile if you want to indicate specific values (note that built-in profiles are read-only).

Most recent interfaces accept 115200 bauds, which is more than enough. Most interfaces will have a *byte size* to 8, *stop bits* to 1, *parity* to *none* and *flow control* to *none*. If unsure, simply try these settings and click on the "Test" button: it will try to connect with the hardware. If it works, then the setup is done! Press OK at the bottom. If it does not, please check the specifications of your serial interface.

Run the client

Now that AT3 is ready, we must run a small "client" Z80 program on your old computer that will receive the data sent by AT3, and play them. The only thing that is sent is the PSG frames, for each PSG (so if your computer supports 6, 9 or more channels, this will work!).

Note: if you're interested in the exchange format, it is available in the package in the players/serial/doc folder.

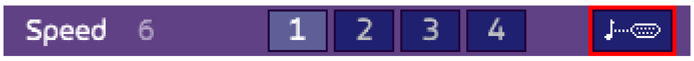
The client can be assembled via Rasm, but as a convenience, the binaries are already present. For example, on CPC, open the DSK located in *players/serial/z80/binary/SerialCPC.DSK*.

There are several files in there, each related to specific serial interfaces, but also the PSG hardware (normal, or PlayCity for example). For example, if you're using the Mini or CPC Booster, run "booster.bin". The same interface with the PlayCity will be "booplayc.bin".

This will show an empty screen with a blue border: the computer is waiting for PSG values! If the border becomes red, it means than an error occurred: please check your hardware and cable! And restarts the client…

Start the fun!

Now that AT3 is set-up and the client is running, starts the communication! Simply click on the "serial" icon at the top-right of the screen:



If any error occurs, a pop-up will show. Else, the icon will change to a little arrow, meaning the communication is working! If at any time you want to stop this, click on the icon again.

Now start composing or playing the song!

All the sound that is emulated is now transferred to your old computer. Muted channels are also muted on the computer. So it really allows you to compose on the hardware the way you would if AT3 was on the old computer!

Please note:

- The sound is NOT muted on your PC/Mac, so both computers emit sound. It is up to you to mute your speakers. Please contact me if you think this is bothersome.
- Serial communication stops when the setup or song properties panels are open, or when a song is loaded.
- For now, the client does not do any buffering, but plays the PSG stream as soon as it receives a PSG frame. At 115200 bauds, it allows real-time without any problem. Please contact me if you think a more sophisticated system should be used.
- Digidrums/samples replay is not supported yet.

Command line tools

Arkos Tracker is not a monolithic software and has many command line tools and options.

Main software

First of all, the main software can be given a song to load:

```
ArkosTracker3 <path to the song to load>
```

If the song is not found, or could not be loaded for some reason, it is ignored (and the reason displayed in the command line).

Tools

In the “tools” folder are all the command line tools. They will allow you to export any song to any format AT3 shown in the export menu (AKG, AKM, etc.). With that, developers can generate a source/binary of the music without having to manually export the AKS song. Whenever the musician produces a new version of their music, a single line will generate the output you need.

Each tool comes with a help if called without parameters, so don’t be afraid :).

The list of tools is as follow. What they are doing is pretty straightforward so I won’t explain what they are doing:

- SongToAkg
- SongToAkm
- SongToAky
- SongToFap
- SongToEvents
- SongToRaw
- SongToSoundEffects
- SongToVgm
- SongToWav
- SongToYm

That’s it!

Players overview

Arkos Tracker 3 comes with several players. Which one to use depends on your needs.

- **AKG**: Your go-to player. Good balance between speed and memory.
- **AKY:** Fast player (12 scanlines on CPC!). But the music are also bigger. Partially superded by FAP (see below).
- **AKM**: Optimized in memory. An interesting alternative to CNGSoft's Chip'n'Sfx. But much more powerful!
- **FAP**: CPU stable, fastest with an excellent compression ratio, by Hicks/Vanity and Gozeur/Vanity! For demos.
- **SE**: Stand-alone **S**ound **e**ffect players. Useful if you don't need music, only sound effects.
- **MOD**: Player for MODule (full-sample).

What player to use?

To make it simple:

- Working on a game? AKG, or AKM (more limited).
- Working on a demo? Test FAP first (CPC only for now), or AKY (multi-platform).
- Working on a sized-limited demo? AKM.
- Need more than 3 channels? AKY.
- Need samples (no PSG sound!)? MOD.
- Need short samples along PSG sounds (i.e. digidrums)? AKY (see this tutorial).
- Need pitched samples along with PSG sounds? AKY in a near future (player is WIP).
- Don't need music, only sound effects? SE.

RAM/ROM players

The RAM players are using self-modifying code as an optimization. This is a drawback for ROM productions, so ROM players have been developed: they use a buffer instead, at a small cost of CPU. See below for the availability.

Hardware availability

Z80 (Amstrad CPC, ZX Spectrum, MSX)

- **All** the players: AKG, AKY, AKM, FAP, SE, MOD.
- **Sound effects** support for AKx, plus SE (stand-alone sound effect player).
- **ROM** version for all the players (except MOD and FAP).
- Support of player configuration to optimize both CPU and memory.
- **Only MOD can play samples so far (see below).**
- **FAP** is currently only supported on CPC, single PSG, no sound effects.
- Supported extensions (**AKY only**):
  - PlayCity (9 channels, Amstrad CPC)
  - TurboSound (6 channels, Spectrum)
  - SpecNext (9 channels, Spectrum)
  - FPGA Psg (6 channels, MSX)
  - Darky (6 channels, MSX)

68000 (Atari ST)

- AKY, without sound effect support, but with an optional **SID** implementation. Player made by ggn, to be downloaded here.

6502 (Apple 2, Oric)

- AKY only.
- Supported machines:
  - Apple 2
  - Oric (both thanks to Arnaud Cocquière)
  - Atari 8-bit with 48K RAM (800 / 800XL / 1200XL / 600XL / 65XE/ 130XE / XEGS) plus the **SONari** extension (thanks to Krzysztof Dudek).

Vectrex

- A AKY player exists for AT3, made by Malban. Check out his repository.

VG5000

- A basic buzzer player *was* done in AT2 with the Lightweight (LW) player. Since AT3 does not have the LW player anymore, I'm waiting for some interest from the VG5000 community before adapting it to the AKM.

Other platform?

- Do you need a support for another platform? Contact me and we may find a way!

Samples support

Samples are not supported by any player for now, except the MOD player (which is 100% sample, no PSG). The reasons are:

- Not many people seem to care. If enough people ask me to do it, I can make a player of course. But be specific about what you exactly need. Remember that, on CPC at least, playing samples takes **MOST OF THE CPU**, unless complicated/cumbersome code is done (interleaving small chunk of code and sample playing), making it highly complicated to produce an easy-to-use and generic player.
- Playing samples along the PSG is tricky on 8-bit hardware, and each coder has its need, so if you want samples, you should code your own sample player.
- Digidrums can be played (sample triggering detection via events) but you have to play the samples by yourself.

Platform interoperability

Can a song exported in a PSG of 1 mHz (like on an Amstrad CPC) sounds well on a Spectrum? It depends mostly on the player, and a bit on the song:

- AKY encodes all periods, software and hardware. So it *will* sound different from a platform to another. You should export the song once again, targeting the right PSG frequency.
- AKG and AKM songs are composed of notes, not period. So in theory, the song can be directly used to another platform. **However,** pitch effects are period-based, so they will sound differently.

Morality: listen to the song on the real hardware, or load the song in AT3 and set the PSG according to the hardware your are using. Correct the pitch effects/tables if needed! So this means that you may need several versions of a song if you plan on targeting several platforms.

The AKG player

The “G” of AKG stands for “generic”, because it is the default player you should use in your production.

It can play all the PSG features of the song, but does not handle samples directly: it is up to you to play them. This player is the best compromise between power and efficiency. The songs and player are optimized both in speed and memory. This is probably the player to use in a game, as it handles long music very well. It also has a support for sound effects. Demos may use it, but cycle-accurate code won’t like it: it can not be stabilized anymore, or with great difficulty, because of the code complexity.

Pros

- Supports all the features of AT3 (effects, subsongs, etc.).
- Best compromise between speed/memory.
- Sound effect support.

Cons

- Can not be stabilized, due to its complexity.
- Rather large player (3kb).

CPU wise, it is hard to indicate how many cycles are spent by this player, but the average music goes from 25 scanlines to 35, on a CPC. This is a bit slower than the AT1 player, but it is also much more powerful. CPU will increase the more effects you add.

This player is quite complex, because of the possibilities of the instruments, plus the columns of effects. This explains why the code is rather large. I sometimes decided to duplicate a few big chunks of code instead of having more generic code: the player would have been dramatically slower.

Limitations

A few limitations are present in this format, but I believe no-one will ever encounter them:

- Only 255 arpeggios or pitches are possible. “This is already the case in the editor”, are you going to say. Right, however, inline arpeggios (effects such as B37 or C37C) generate one arpeggio (for each distinct value of course), so hugely complex music may generate too many arpeggios.

- The generated effect blocks can only be 64k max. This is insanely big, so don't ever pretend to break this limitation, I won't believe you.

Can I optimize in speed?

The way your song is built has a huge impact on the CPU. Most of you won't have to bother about it, but if your coder asks for a few CPU-cycles to save, here is what you can do for him.

Effects, though very handy, takes much more than using their counterpart within the instrument:

- For example, using the Arpeggio inside an instrument doesn't cost much: using the Arpeggio Table inside a track costs more.
- The same for the Pitch Table.
- Glide effect costs many cycles! Prefer the simpler Pitch.
- The Reset effect is handy, as it resets all the effects. However, many flags must be reset, making it also quite costly. If you can, reset only the effects that needs to be reset. Most effect have their "reset" counterpart:
    - Use A00 to stop the Arpeggio Table.
    - P00 to stop the Pitch Table.
    - Vf to set the volume to max.
- However, it is faster to use one Reset effect instead of two specific reset effects.
- Use the player configuration feature!

The AKM player

The **AKM**, or Minimalist player, is a small but very powerful player, especially useful for 4K demo, or 64kb games, for example.

It is an interesting alternative to CngSoft Chp’n’Sfx, which has a really small, albeit limited, player. AKM is much more powerful, but also bigger.

Pros:

- Smaller player and songs.
- Sound effects managed.

Cons:

- Slower than the other AT2 players.
- A bit less features.

Here are the limitations regarding what this player can accomplish:

- Only one PSG per Subsong.
- No “hard to soft” sounds.
- SoftAndHard more limited: hardware part is only with fixed hard period (allows “Ben Daglish” effects).
- No events.
- Only speed change at the start of a pattern are encoded.
- Arpeggio and Pitch values in the Expressions are limited in range: from -64 to 63.
- Arpeggio and Pitch Expressions are limited in size: 127 items only.
- No Legato.
- In an instrument, period in Software mode must be automatic, cannot be forced.
- No retrig.
- Effects:
  - Reset with volume
  - Set volume
  - (fast)Pitch up/down
  - Arpeggio table (inline will be converted)
  - Pitch table

    - Force Instrument Speed
    - Force Arpeggio Speed
    - Force Pitch Speed

On exporting, warnings are shown if some song items could not be correctly exported.

The AKY player

> The AKY has been partially superseded by the FAP player, depending on your needs and constraints. Compare both before making your final decision!

The **AKY** player is some kind of YM-like format, but more clever :). Basically, the YM format (invented by Leonard/Oxygen) is a stream of the values sent to the PSG, frame by frame. Obviously, it takes quite a lot of space (about 45 kb per minute of music). The biggest advantage of such format is that, once decoded, playing a frame is trivial and fast.

A LZH compression is then added to reduce the size to a few kilobytes. Sadly, decoding it in real-time takes too much resources for most 8-bit computers.

Various formats exist to try to find a good compromise between memory and speed (AYC by Madram on CPC, MYM on MSX), with the raw YM stream as an input. Well, AKY is most of the times, much better than them :).

Basically, AKY relies on the structure of the song to determine when tracks and notes start and end, and try to code sub-sequences into bigger sequences, optimizing what is needed to be encoded.

Pros

- Short and very fast player.
- Contrary to the aforementioned YM alternatives, AKY does NOT relies on any buffer! Only the song and the player have to be accounted for.

Cons

- Bigger song than with the AKG/AKY players (i.e. players that don't rely on a stream).

Inaccuracy?

A small technical explanation: in the AKY format, the PSG register blocks are stored in an optimized way, only differences are encoded. The problem is that, in some cases, some shared values between channels (such as noise/hardware envelope/period) can be considered “the same” whereas they are not (one channel modifies the value, but the second channel considered it “equal, not changed”). As a result, the player might use the first value and not the second, which it should. Most of the times, only a few frames are different so you will probably not hear the difference.

So to sum up, the problems will arise only if you change the noise in two or three channels at the same time, in close vicinity (or you use 2 or 3 hardware sounds at the same time).

Such corner cases are not corrected, because neglecting them allows for even more speed.

How to make an optimized AKY

If you are working on a demo and you know you need a fast player, please read (or have your musician read) the following to have an efficiently built AKY song.

- Tracks that **sound** the same are optimized away. I emphasize on “sound”. Tracks may be the same in your song, but may sound differently (for example, an effect can affect the second iteration of the track (a change of volume, an arpeggio, etc.)). Make sure your tracks are as “unique” as they pretend to be: reset all the effects at the beginning of each track.
- Avoid long and not repeated sequences, such as long slide. Each frame being different, they have to been fully encoded and the odds they can be reused is thin. Vibratos are fine.

FAP player

FAP is a brand new player by Hicks/Vanity and Gozeur/Contrast. Check its repo here.

Of course it has been integrated in AT for a seamless use!

Pros

- Blazing fast. About 10 scanlines on CPC, depending on the music!
- CPU-stable. Very useful when working on a demo!
- Excellent ratio (to my dismay, beats AKY most of time. But the war is not over!).

Cons

- For now, CPC only.
- Only 3 channel. If you need more, check the AKY player.
- Not Disark-ready yet, so it assembles with RASM only, unless you use the binary blob provided in the original repository.
- No sound effect support.
- A 3 kb buffer is required. Take that in account when comparing with the AKY, which is buffer-less.
- No ROM support.

This player clearly targets demos. You might want to use it for a game, but the memory footprint would probably be too high, plus there is no sound-effect support.

The MOD player

Arkos Tracker 3 can play PSG and Samples indifferently. However, playing samples on 8-bit machines is always a problem as it takes most of the CPU.

Currently, **only a CPC Z80 player is done** (but it's actually easy to adapt to any other machine. Once again, ask me!).

The MOD player allows you to play a 3-channel music with a few effects:

- Pitch up/down.
- Arpeggio table.
- Arpeggio 3 notes.
- Arpeggio 4 notes.
- Reset.

The other effects will be ignored, no need to remove them from the song.

Exporting a music

Once you have your 3-channel music, you will want to export it. There is no "Export as MOD" option. Instead check for the "Export as RAW". Why? Because "raw" is a binary format in which you can encode whatever data you want to export. Its simplicity makes it easy to parse. As playing samples costs a lot of CPU, the raw format fits well our needs.

The raw export seems a bit overwhelming at first. **Only the following must be ON**. Set the other check-boxes to OFF (this is also explained in the PlayerMod source)!.

- Encode Song/Subsong metadatas.
- Encode Reference tables.
- Encode Speed Tracks.
- Encode instruments.
- Encode effects (even if you don't need them).
- Encode arpeggios: only if you use them.
- Encode heights in linker.
- Pitch Track ratio: 0.25 (but this is an approximation).

As for the samples:

- “Export used samples?” must be ticked.
- The sample MUST be encoded with an “offset” of 128.
- The “amplitude” should be between 8 and 10, it’s up to you to choose what is best for the song.
- You MUST use a “padding length” of 320 at least, but use 450 or more if you experience strange problems, such as the sound stopping (technically: at least equal to “PLY_MOD_IterationCountPerFrame” + 1, else strange things will happen). If not, the player will read the data past the sample data, reading the header of the next instrument, sending wrong data to the PSG (at best, cutting the sound). Increasing to 450 or more is needed if you use the upper notes.

And that’s it.

Two warnings:

- **Do not use any PSG instrument** (use Tools > Optimize instruments to remove them), the player does not know how to play them. The first “empty” instrument remains present, this is nomal.
- **Make sure your song does not fill the memory up!**

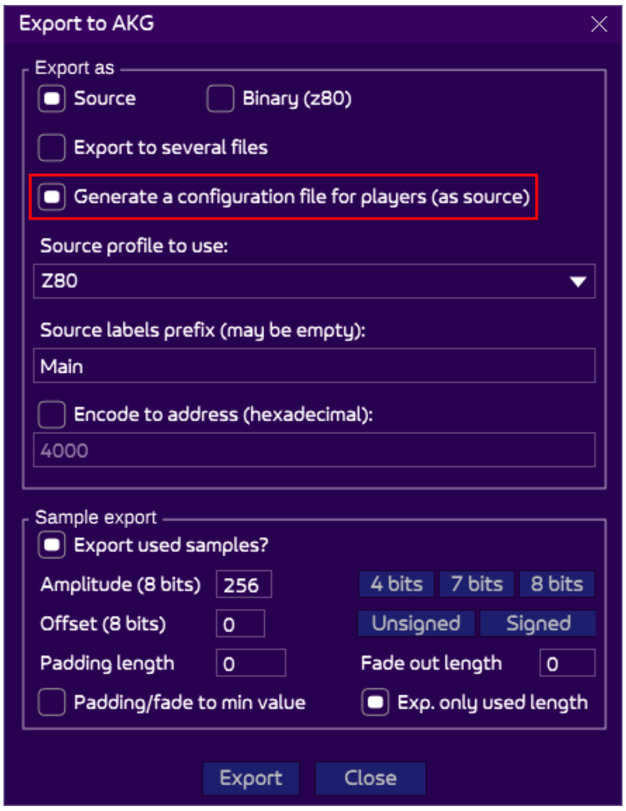
Load your song along with the player and enjoy!

Player configuration

Imagine you have a song that doesn't use 100% of the AT features. Do you think it is fair to use a player which wastes memory (and probably CPU) to manage these anyway? Sure, you could delve into the source and remove these parts of code by yourself. But it's tedious, error-prone, maybe you don't know anything about players, and most of all, you don't have time to do it properly, you have a production to finish!

Well the good news is that **Player Configurations** will do the job for you.

An option appears when exporting a song or sound effects: "Generate a configuration source for players".


Export to AKG

Export as
[✓ Source] Binary (z80)
Export to several files
[✓ Generate a configuration file for players (as source)]
Source profile to use: Z80
Source labels prefix (may be empty): Main
Encode to address (hexadecimal): 4000

Sample export
[✓ Export used samples?]
Amplitude (8 bits) 256
Offset (8 bits) 0
Padding length 0
Fade out length 0
Padding/fade to min value
[✓ Exp. only used length]

If activated, a source file will be generated along your music. It will have the same name, post-fixed with the name "_playerconfig". If you open the file, it will look like this:

```
PLY_CFG_ConfigurationIsPresent = 1
PLY_CFG_UseSpeedTracks = 1
PLY_CFG_UseEventTracks = 1
PLY_CFG_UseTranspositions = 1
PLY_CFG_UseHardwareSounds = 1
```

```
PLY_CFG_UseEffects = 1
PLY_CFG_SoftOnly = 1
PLY_CFG_SoftOnly_Noise = 1
PLY_CFG_SoftOnly_ForcedSoftwarePeriod = 1
...
```

What is that? Very simple: AT has analyzed your song and has generated all these flags to indicate the player (whatever it is) that Speed Tracks are used, so are Event Tracks, and so on.

The player *might* check these flags and assemble parts of its code or not according to these flags! So it will take less memory, less instructions to execute, which can also lead to substantial savings in CPU.

Why *might*? Because, depending on the export format, these flags may not be relevant to the player. For example, the AKY player, being a “streamed” player, does not care about technicalities such as “use forced software period in Software sound”. However, AKG and AKM players will find these very useful information, as it means one branch of code will never be executed, so the whole test about the “forced software period” and its subsequent code will not be compiled.

How to use it? Very simple. As it is done in the testers, simply include the Player Configuration file(s) **BEFORE** the player itself is assembled. This is important to do it before, because the player checks the presence of these flags at its beginning. Example:

```
org #1000
...
include "MySong_playerconfiguration.asm"    ;Do that BEFORE including the
player!
include "PlayerAky_CPC.asm"
include "MySong.asm"
```

Technical note: the players test the flag *presence*, not the flag *values*.

Can all the players use these Player Configuration files?

The main ones are (AKG, AKY, AKM plus Sound Effects) for Z80. What is missing, for now, are the specialized ones (AKY for Turbosound or PlayCity, AKY stabilized). Please tell me if this is needed, and I will do it.

I have three songs, and one player… How to handle this?

Simply include the three Player Configuration files one after the other **BEFORE** the player. The flags will stack up! Example:

```
include "FirstLevelSong_playerconfiguration.asm"
include "SecondLevelSong_playerconfiguration.asm"
include "GameOverSong_playerconfiguration.asm"

include "PlayerAky_CPC.asm"

include "FirstLevelSong.asm"
include "SecondLevelSong.asm"
include "GameOverSong.asm"
```

Note: you should use one song and three subsongs instead of having three songs. You can share their instruments/arpeggios/pitchs and save a lot of memory!

I have sound effects, does it work too?

I got you covered. Simply export a Player Configuration for them too, include it besides the Player Configuration of the song, **BEFORE** the player, and voilà!

This is too complicated for me! I don't care about optimizations and all that! Can I ignore all this?

Yes! If you look at the beginning of the code of the player, you will see this:

```
IFNDEF PLY_CFG_ConfigurationIsPresent
      PLY_CFG_UseHardwareSounds = 1
      PLY_CFG_UseRetrig = 1
      ....
ENDIF
```

This basically means that if the "player configuration" is not there (because Player Configurations have not been added, because you don't care about these), flags will be automatically set. As a consequence, the full player is compiled. But don’t complain if it takes too much memory or CPU :).

How much will I gain?

It depends on the player and on your song.

You won’t save much with AKY, since the code is quite simple and has not many branches. You could save a hundred of bytes, a few dozen of CPU cycles.

However, the AKM and the AKG especially will gain a lot from Player Configurations! You can save up to one and a half kilobyte on the player, and a few scanlines of CPU!

So don’t hesitate, use this feature and don’t go back.

Events

Events are “signals” written at specific moments in your song, which the player can detect. As explained here, this is useful to synchronize your production with a music without the coder having to count the frames. A demo can be music-driven!

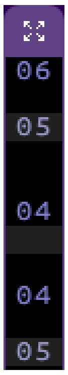
Events are an 8-bit integer from 0 to 255 (FF in hex). What it means is up to you. You could simply use `01` and say to the coder “`01`, or any other number, means the demo moves forward”. Or something more sophisticated like:

- `01` means the colors must flash.
- `02` means the screen must wobble.
- `03` means the dots effect must start.
- `04` means the 3D effect must start”.

This is entirely up to you to know what these numbers mean.

Writing events

This is explained in the aforementioned page, but here is a summary. At any times in event tracks, write a number between `01` and FF.



```
06
05

04

04
05
```

Nothing happens in AT when writing these. Only the players on the hardware can receive and interpret these.

Receiving events

This is where this get technical. Not all players can naturally handle events. AKG does, but AKM and AKY does **not**. Don't worry, there is a trick for it explained below.

Export the song as you would normally do. It contains the events within its data.

In your assembly code (Z80 for example), play the song normally as it is done in the testers provided in the package:

```
    ;Main loop. I suppose this runs at 50 Hz.
    call PLY_AKG_Play

    ;Is there any event? 0 means "no effect".
    ld a,(PLY_AKG_Event)
    or a
    jr z,NoEvents

    ;Ah, there is an event!
    cp 1
    jr z,FlashColor
    cp 2
    jr z,WobbleScreen
    ;... and so on.

NoEvents
    ;Continue your main code here...
```

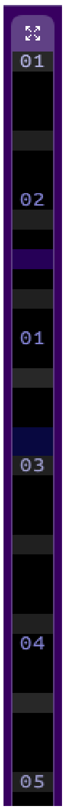
This is only a raw code but it is a starting point (if you handle many events, it would be better to have a pointer table and jump directly on the code instead of chaining many CPs, but optimization is not the purpose of this tutorial!).

This is it if you use the AKG player. If using any other player, you might be embarrassed as none supports events directly. Read on…

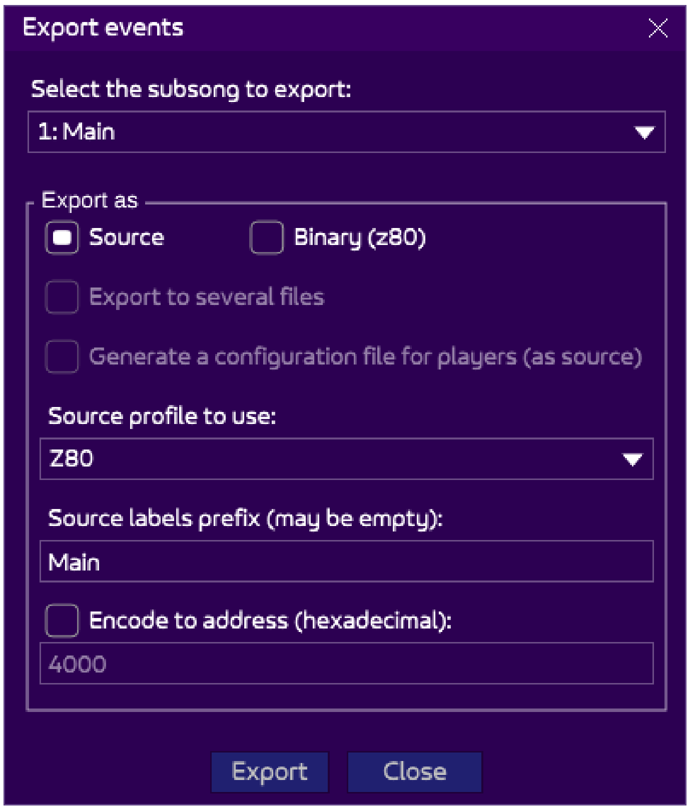
Exporting events

If not using AKG player, you will have to handle the events by yourself. Don't worry it is easy. Also, the AKY digidrums player include a parser of such events, so you can rip its code.

Let’s pretend that your events were written as this in your song (with a speed of 06).



First of all, export the events. Go to File > Export > Export events. A dialog opens. For our example, make sure that the source profile is Z80, and source. Then press Export:


Export events

Select the subsong to export:
1: Main

Export as
[✓ Source] Binary (z80)
Export to several files
Generate a configuration file for players (as source)

Source profile to use:
Z80

Source labels prefix (may be empty):
Main

Encode to address (hexadecimal):
4000

A Z80 source code is generated. Let’s look at it (this example has been cleaned of some comments and labels for cleanliness):

```
; Events generated by Arkos Tracker 3.

MainEvents

MainEvents_Loop
    dw 1    ; Wait for 0 frames.
    db 1

    dw 42    ; Wait for 41 frames.
    db 2

    dw 42    ; Wait for 41 frames.
    db 1

    dw 36    ; Wait for 35 frames.
    db 3

    dw 54    ; Wait for 53 frames.
    db 4

    dw 42    ; Wait for 41 frames.
    db 5

    dw 168    ; Wait for 167 frames.
    db 0

    dw 0    ; End of sequence.
    dw MainEvents_Loop    ; Loops here.
```

This is pretty straightforward. It is only a list of pair of values:

- First, a 16-bit number representing *how many frames to wait -1*:
  - 50 means wait for 49 frames before reading the event number.
  - 49 means wait for 48 frames before reading the event number.
  - …
  - 2 means wait for 1 frame before reading the event number.
  - 1 means “don’t wait”: read the event.
  - 0 means “end of list”, followed by the address where to loop to. Continue the parsing.
- Then, a 8-bit integer of the event number. If 0, it means *no event*, simply continues the parsing.

After reading a valid event (>0), you can stop the parsing for this frame.

In the unlikely event that more than 65535 frames must be waited, and thus a 16-bit number is not enough, then more than one wait is encoded, with an event number of 0 (meaning *no event, continue parsing*). There is actually no special code to perform if you followed the rules described above.

There is no code provided to parse this, I will leave it to you (unless I am asked to do it!), as it is very simple.

A simple Z80 example

If you're working in Z80, this page is for you. There are already ready-to-use testers in the AT3 package ("*players/playerAkg/sources/tester*", to name only one), but here is a very simple example of how the players work.

> Garvalf has also created working examples (for CPC, Spectrum and Garvuino), which you can find on his GitHub repository (for AT2, but it works the same). Thanks to him!

We will use AKG player here, which is the "generic" player, but the other players work the same way.

This page is made to be straightforward, so let's be concise.

Export of the song via command line

Let's use the command line tools to export your song. You can do it from the graphic interface inside the software, but we want to go fast!

There are many example songs, but let's focus on "*Targhan – A Harmless Grenade.aks*" located in the *songs|ArkosTracker2|3Channels* folder.

The command line tools are located in the *tools* folder. The following command will create the source code of the exported music, which we call *music.asm*. The input file is of course the music we talked about just above:

```
SongToAkg.exe "Targhan - A Harmless Grenade.aks" music.asm
```

Assemble the whole

The AKG player is located in the *players|playerAkg|sources* folder, and named *PlayerAkg.asm*. Let's create a wrapper around it to play the song every 50hz. We use Rasm to assemble. If you want to use any other assembler, please check this tutorial.

This sample targets the Amstrad CPC, but choosing a MSX, Spectrum or anything else works exactly the same. The only specific code is the wait for the frame flyback. Please adapt it according to your target computer.

```
    org #1000

    ;Moves the system out of the way.
    di

    ld hl,#c9fb
    ld (#38),hl

    ;Put the stack out of the way (it can destroy the beginning of our
code, we don't care).
    ld sp,$

    ;Let's initialize the song.
    ld hl,MusicStart    ;Declared below.

    xor a               ;Subsong 0 (the main one).
    call PLY_AKG_Init

    ;Wait for the frame flyback.
    ;On CPC, we do it this way:
Sync:
    ld b,#f5
Sync2: in a,(c)
    rra
    jr nc,Sync2

    ei
    nop
    halt
    halt
    di

    ;Let's play our song!
    call PLY_AKG_Play
    jr Sync
    ;Of course, we need to include the player and the music to play.
    ;Uncomment the target you want. Declare this BEFORE the player is
included.
    PLY_AKG_HARDWARE_CPC = 1
    ;PLY_AKG_HARDWARE_MSX = 1
    ;PLY_AKG_HARDWARE_SPECTRUM = 1
    ;PLY_AKG_HARDWARE_PENTAGON = 1

    include "PlayerAkg.asm"
MusicStart
    include "music.asm"
```

That’s it! Just execute this code, starting in 0x1000 and music will be heard!

Using a song in a production, using Rasm

First, we will study the simplest case : using Rasm. Don't worry, using other assemblers will be covered in the next part. However, you must read this part first to understand how the player works, then you can switch to the next part where we'll take care of *your* assembler *problem*.

So! I consider four files were generated:

- Grenade.asm (the song)
- Grenade_playerconfig.asm (the optional config file for optimization)
- SoundEffects.asm (the sound effects).
- SoundEffects_playerconfig.asm (the optional config file for the optimization of the sound effects).

We will use the AKG player, but any other would work the same way.

You *could* open the *players/playerAkg/sources/tester* folder and pick up the tester that would fit, and that would be it! But as it is a tutorial, I'll explain everything from scratch.

Let's make a code

Our goal is to make our own code to play the song.

Its structure will be very simple:

1. Initialize the code
2. Initialize the song
3. Wait for the frame flyback (explained later)
4. Play the song
5. Go back to 2!

Initialization

We will start with a ORG to define where our code starts, and various initialization, which are platform depend. You can check the testers directly on how this is done, but we'll have a quick overview here.

On CPC, we can start as low as #1000 (or even lower, but watch out for your Basic program!), and we "kill" the system by putting the *DI : RET* instructions in #38. This will prevent the system from messing with our code.

```
    ;CPC:
    org #1000
    di
    ld hl,#c9fb   ;DI, RET
    ld (#38),hl
```

On MSX, you need to declare a small header for the BLOAD command to work, it's easy (more info here):

```
    ;MSX:
    org #b000
    db #fe     ;Declares a binary header.
    dw TesterStart    ;Start of the tester, just after this header.
    dw TesterEnd      ;End of the tester.
    dw TesterStart    ;Execution address.
TesterStart           ;Starts our real code here.
```

Don't forget to add the *TesterEnd* label at the *very* end of the source, even as we start adding more code (do **not** put anything after!).

On Spectrum/Pentagon, it seems we only have to do this:

```
    ;Spectrum/Pentagon:
    org #8000
    di
```

Initialize the music

Now that this technicality is done, we can initialize the music. All the players work the same way: we call the "init" method of the player and gives it the address of the music.

One question we might ask is: where are the music and the player? Don't worry, we'll include them later.

```
    ld hl,Music             ;The address of the music.
    ld a,0                  ;What subsong to play (starts at 0)?
    call PLY_AKG_Init       ;Call the init method of the player.
```

(Yes, "*ld a,0*" can be optimized to *xor a*, I didn't want to scare the beginners). The subsong "0" is in fact the "1" in the AT2 editor. In assembler, we think in "index", which starts at 0. If later, you want to play another subsong (for example, a "game over" subsong), simply call the *PLY_AKG_Init* method again with the corresponding subsong index, and there you go.

This "init" method is always named the same way in every player, only the "AKG" part changes. For example:

- For the AKY player, the init method will be PLY_AKY_Init
- For the Lightweight player, PLY_LW_Init
- and so on.

The only difference between players is that AKY doesn't manage subsongs (there is only one), so there is no need to set A. To know that, simply check the testers of these players to check how to use them.

…So, we have initialized the player. Now it knows about where the music is, and what subsong to play. Note that this is a mandatory step!

Initialize the sound effects

Maybe you want some sound effects? Now is the right time to initialize them too. Just like with the music, it is a mandatory step, and it must be done once. Simply call the init method of the SFXs, giving it the address of the sound effects:

```
    ld hl,SoundEffects          ;Address of the sound effects.
    call PLY_AKG_InitSoundEffects
```

Once again, we will include the sound effects later, so don't worry if the program doesn't compile yet.

Frame flyback

I told before about this "frame flyback" stuff. On most retro machines, music, but also animations, are synchronized with the frame flyback. This happens 50 or 60 times per seconds (thus, 50/60 Hz). At that moment, the electron canon that displays the image on the screen is at the top, and will move towards the bottom to display a frame.

We synchronize the music and sound effect to the moment the canon is at the top. By doing this, we can assure the music is played at constant rate, and at a decent one (the faster, the more notes we can play!). 99% on the songs are expected to be played at 50 Hz, but you could also want a song that plays at 100, 150, or even 300 Hz! On the opposite, 25 or 12.5 Hz are also possible and can still give good results. Just know that AT3 can handle all these! But for now, let’s stick to our good old 50 Hz.

Waiting for the frame flyback is once again system dependent:

```
Sync
    ;CPC:
    ld b,#f5
    in a,(c)
    rra
    jr nc,Sync + 2
    ei
    nop
    halt
    halt
    di

    ;MSX / SPECTRUM / PENTAGON:
    ei
    nop
    halt
    di
```

Please only use the snippet your platform requires. After this, we can finally play one frame of our song.

Play the music

This is very simple:

```
    call PLY_AKG_Play
```

That’s it! As the player has been already initialized, this is the only thing to do. No need of any parameter.

Note that we only played one frame of the music. Now we need to create a loop for the music to be played *ad vitam eternam*. This is simply done by adding a jump to the frame flyback code, that we (cleverly) marked with a “*sync*” label:

```
    jp Sync
```

That’s it! We have a music that plays. We could stop here, but maybe you want, at some point, to stop the music.

Stopping the music

This is done as simply as playing it:

```
    call PLY_AKG_Stop
```

Where to call this? Well, you could add a keyboard test after the “play”, and if a certain key is pressed, go to a subroutine that makes the “stop” call. This is very platform-specific, so I won’t enter in these details. You should be able to do it by yourself!

Please note that once the music is stopped, you should **NOT** make a call to the Play method. The Stop method **only** cuts the PSG registers, thus stopping all sound. But it does NOT prevent the player from playing if called again.

Play the sound effects

What about adding sound effects? If you have initialized them, as seen above, playing one sound effect is simple. All has been explained here, but here a little summing up anyway. Call this to play the first sound:

```
    ld a,1 ;Sound effect number (>=1)
    ld c,0 ;channel (0-2)
    ld b,0 ;Inverted volume (0-16)
    call PLY_AKG_PlaySoundEffect
```

When to call this? This is highly dependent on your system, but you could make a keyboard test to play a sound.

Note: calling this method will only *program* the playing of a sound effect. It is the *PLY_AKG_Play* that will play it along with the music, frame by frame.

Stopping a sound effect

This is done by calling a simple method, plus the channel where the sound effect is. So this does not actually stop one specific sound effect, but any sound effect that is on the selected channel.

```
    ld a,0   ;Channel (0-2)
    call PLY_AKG_StopSoundEffectFromChannel
```

Declare the players and music

Last part! And probably where all the subtleties are. Our code does not compile yet because we didn't include the player, music, and sound effects files! Let's do that, then. Remember, we have already generated all these files in the previous part of this tutorial.

At the bottom of the source, include the music:

```
    include "Grenade.asm"
    include "Grenade_playerconfig.asm"   ;Optional.
```

Note that we also included the player configuration file. You can do it anywhere (or not at all, this is optional!) but it must be put **before** the player, else the player can not know how to configure itself, and will use default parameters (so no optimizations will be performed, which is a shame!).

Now include the sound effects, if you're using them:

```
    include "SoundEffects.asm"
    include "SoundEffects_playerconfig.asm"   ;Optional.
```

Once again, we used the player configuration of the sound effects, for more optimization.

Finally, let's include the player. But warning! AT3 is multi-platform, so the player must be told what platform to use. CPCers are lucky, they don't have to do anything, as CPC is default. Other platforms must declare their identity. This is done with a single line, don't worry!

Also, we need to indicate another flag if you want to include the sound effect support. Once again, a single line is needed.

```
    ;Selects the hardware target:
    ;PLY_AKG_HARDWARE_CPC = 1   ;Declare CPC platform. Default, not
needed!
    PLY_AKG_HARDWARE_MSX = 1    ;Declare MSX platform.
    ;PLY_AKG_HARDWARE_SPECTRUM = 1    ;... and so on.
    ;PLY_AKG_HARDWARE_PENTAGON = 1

    ;Sound effect support? Then uncomment this:
    ;PLY_AKG_MANAGE_SOUND_EFFECTS = 1

    include "PlayerAkg.asm"
```

Note that unused lines **must be deleted or commented**. Do **NOT** put 0 instead of 1, it will not work! The players test the variables *presence*, not their *value*!

And that’s it!! If you did things right (!), you should hear music (and sound effects if you plugged things right).

What order?

I included the music before the player, but this is not mandatory at all. It is actually more intuitive to, instead, include the player first, then the music and the sound effects. If you do so, remember to include the player configuration files **before** the player, which may be counter-intuitive, because the music/sound effect files they are related to will be included later. Like this:

```
    PLY_AKG_HARDWARE_MSX = 1
    PLY_AKG_MANAGE_SOUND_EFFECTS = 1

    ;Always include the player configuration files BEFORE the player.
    include "Grenade_playerconfig.asm"
    include "SoundEffects_playerconfig.asm"

    ;This is the player.
    include "PlayerAkg.asm"
    ;Music and sound effects.
    include "Grenade.asm"
    include "SoundEffects.asm"
```

This should get you started!

You can assemble your code by typing:

```
rasm <MySourceFile.asm> -o Test
```

(of source, replace *MySourceFile.asm* with the name of the source you just created. "*Test*" is the base name of the generated file. In our case, it will generate *Test.bin*).

And of course this should assemble without any error.

Quick testing with an emulator

You can inject this code directly in an emulator for testing, for example. On CPC, open **Winape**, pause it (F7). The debugger opens. At the bottom, select "Any" in Memory. In the Monitor just above (where all the hex numbers are), right-click and select Goto… Enter 1000 (this is our ORG value). Then right-click again on the monitor and Load. Select your Test.bin file. Click on Play in Winape, and under Basic, type call &1000. You should hear some music!

What now?

If you use Rasm, you can stop reading here. If you are using another assembler, then go on to the next part!

Using a song in a production, using any assembler

Sooo, you don’t want to use Rasm in your production, but you’re a bit embarrassed because the player and music sources are not compatible with your assembler. Fortunately, I got you covered.

Thanks to Disark, you will be able to convert the sources into ones that your assembler will like (to be more precise: it will convert a *binary* into a source). This command-line tool can be downloaded in its website.

Prerequisite:

- I assume you have read the previous part of the tutorial, because I will not explain everything from scratch.
- You still need to download Rasm to compile the sources. Don’t worry, it is cross-platform! Do it now, thanks.
- You have successfully exported a song (and sound effects optionally) in the AKG format (as an example), so have these files:
    - Grenade.asm (the song)
    - Grenade_playerconfig.asm (the optional config file for optimization)
    - SoundEffects.asm (the sound effects).
    - SoundEffects_playerconfig.asm (the optional config file for the optimization of the sound effects).

For this tutorial, we will emphasize on SDCC, because it is quite used on our 8-bit community, and it is also the most bothersome to care about! But if you’re using Maxam, Winape, Pasmo, Vasm or anything else, it’s going to be even easier!

If you want to follow this tutorial using SDCC:

- I used the version 3.9.0 (which is quite old now). If you have any problem with the latest version, please let me know.
- SDCC requires us to use hex2bin to generate the final binary. Download it here (windows exe here).

Also, we still use the AKG format. Using others (AKM, Lightweight, AKY) is possible and works exactly the same.

Converting your sources

A mandatory step is to convert the player/song/sound effects sources into another source that your assembler will understand. **This must be done only once**, so don't worry. Of course, if you change your songs, you will have to do this step once again!

What I advise is to compile all these into one single file. Most productions don't need more flexibility than this: you probably don't need to separate the player from the song, at first (but if you do, this can be easily done).

Let's create an asm file that will gather all these sources. We will call it *CompileAT3Files.asm*:

```
;Compiles the player, the music and sfxs, using RASM.
;No ORG needed.

;This is the music, and its config file.
include "Grenade.asm"
include "Grenade_playerconfig.asm" ;Optional.

;This is the sfxs, and its config file.
include "SoundEffects.asm"
include "SoundEffects_playerconfig.asm"  ;Optional.

;What hardware? Uncomment the right one.
;PLY_AKG_HARDWARE_CPC = 1
PLY_AKG_HARDWARE_MSX = 1
;PLY_AKG_HARDWARE_SPECTRUM = 1
;PLY_AKG_HARDWARE_PENTAGON = 1

;Comment/delete this line if not using sound effects.
PLY_AKG_MANAGE_SOUND_EFFECTS = 1

;This is the player.
include "PlayerAkg.asm"
```

Note that, in this example:

- I used the MSX hardware. Change the flag according to your hardware.
- I enabled the sound effects.
- I used both configuration file for song and sound effects, to have a fully optimized player.
- The player has been added at the end, to make sure all the flags were set for the player to be well configured. You don't have to keep the same order, you can put the music/sfx after the player. **BUT** the flags and configuration files **MUST** be declared **before** the player, else it won't be able to use them.
- Feel free to change all the parameters to satisfy your needs, of course.

Assembling with Rasm

We are now going to assemble this source file with Rasm.

"But hey, I thought we were converting sources, I don't want no stickin' binary!" you say in a hoarse voice. Well don't worry, we will get a source back thanks to Disark! First, let's compile this with Rasm:

```
rasm CompileAT3Files.asm -o UniversalAt3Files -s -sl -sq
```

This assembles the source above and produces two files:

- *UniversalAt3Files.bin*: the binary.
- *UniversalAt3Files.sym*: the symbol file.

Regenerating sources

Disark will require at least the binary, but shines when given the symbol file: it will be able to reproduce the source in a faithful way!

```
Disark UniversalAt3Files.bin At3Files.asm --symbolFile
UniversalAt3Files.sym --sourceProfile sdcc
```

Here, we have chosen "sdcc" as the source profile, but you can ask for other assemblers. Check here for more info about the arguments.

What matters is that a new file, *At3Files.asm*, has been generated. You can open it: it contains the music, the sound effects and the player, in a syntax that can be understood by SDCC! Isn't that great? And since it is a well-generated source, you can relocate it wherever you want, and access its labels!

Building a tester

All is well, but it is now time to build a tester to check if what I'm selling you (for free) is really working. We're still using SDCC because it's soooo bothersome, but adapting to any other assembler is very easy.

We will embed Z80 code into a C program, but will not enter a single C line (ok… just one). Indeed, SDCC being a C compiler first, Z80 areas must be declared. Let's create a *TesterSdcc.c* file. For now, type this:

```
void main()
{
__asm
        .area tester (ABS)
        .org 0x4000
__endasm;
}
```

This simply declares an asm area (called "tester"), with an ORG at &4000. If you're using any other assembler, only the ORG instruction is useful (in an .asm file, not .c)!

I want you to first test that this compiles with SDCC:

```
sdcc -mz80 --no-std-crt0 --vc TesterSdcc.c
```

This should generate a few files we don't care about for now, but most of all, there should be no error and no warning! If there are, check you code and update SDCC.

Code initialization

Every code in every platform needs a bit of setup to satisfy the hardware/the system (stop interruption, etc.). We covered that in the previous part, but I'll make an example here anyway. We'll choose the MSX as an example because, just like SDCC, it's the most bothersome :).

```
void main()
{
__asm
    .area tester (ABS)
    .org 0xb000
    ;Header of a MSX binary file.
    .db 0xfe     ;Declares a binary header.
    .dw TesterStart    ;Start of the tester, just after this header.
    .dw TesterEnd      ;End of the tester.
    .dw TesterStart    ;Execution address.

TesterStart:           ;Starts our real code here.
    ;... We will have to put our code here.
TesterEnd:

__endasm;
}
```

Ok, we can still compile this without error. This code makes it possible, on MSX, to load and execute this file using the BLOAD Basic instruction.

Add music

Just like we did in the previous part of the tutorial, we'll play music first and see if it works. Reminder:

1. The player is initialized by giving the address of the music, and the index of the subsong to play (0 most of time).
2. We wait for the frame flyback (hardware dependent!) to sync our music on.
3. We play one frame of the song.
4. We loop to the frame flyback label to play our song indefinitely.

```
...
TesterStart:
    ;1 - Initializes the music.
    ld hl,#AHARMLESSGRENADE_START    ;The music.
    xor a     ;The Subsong to play (>=0).
    call PLY_AKG_INIT

    ;2 - Wait for the frame flyback (MSX/Spectrum/Pentagon specific).
Sync:
    ei
    nop
    halt
    di
    ;3 - Play one frame of the song.
    call PLY_AKG_PLAY

    ;4 - Loop!
    jr Sync

    ;Of course we have to include the music/sfx/player!
    .include "At3Files.asm"
TesterEnd:
...
```

Note: you may want to know how I found the AHARMLESSGRENADE_START label. You might have a different result, because it depends on how you generated your song. The first part is the "source prefix" asked by AT3 in the Export panel. But why the upper case? Because Rasm always treats its labels as upper case, and as Disark used the symbols exported by Rasm, logically, all the labels of our "At3Files.asm" are upper case.

This looks like a working tester! But let's compile it. This is done in two passes with SDCC. If you're using any other assembler, this is probably done in one:

```
sdcc -mz80 --no-std-crt0 --vc TesterSdcc.c
hex2bin TesterSdcc.ihx
```

The first line generates a few files, including an IHX file which contain more or less the binary, in a special format. The second line converts it to raw binary. At the end, and if no error occurred, you should have a *TesterSdcc.bin*!

First test on hardw… emulator

Testing it once again heavily depends on your hardware. You can include the binary in a virtual disk and run it on hardware or emulator. For the sake of completeness, I'll test this on a MSX emulator.

I am no MSX expert, but a simple way of doing it is as follows:

- Use MSX Disk Manager (Windows) to create a DSK (default parameters are fine), drag'n'drop *TesterSdcc.bin* in it. Rename it to *Tester.bin* inside the software, else the long name will be bothersome. Save the DSK.
- Use BlueMsx (Windows). Drag'n'drop the DSK, the MSX will boot. Once the prompt is shown, type:

```
bload"tester.bin",R
```

The file will load and the music will be heard! Isn't it great?

Add sound effects

One last step is to add the sound effects. We have already included the sound effects themselves, the player is configured to play them (via the PLY_AKG_MANAGE_SOUND_EFFECTS flag), so there isn't a lot left to do!

First thing, initialize the sound effects. This is exactly the same as for the music:

```
...
TesterStart:
    ;1 - Initializes the music.
    ld hl,#AHARMLESSGRENADE_START    ;The music.
    xor a     ;The Subsong to play (>=0).
    call PLY_AKG_INIT

    ;1b - Initializes the sound effects.
    ld hl,SOUNDEFFECTS    ;Address of the sound effects.
    call PLY_AKG_INITSOUNDEFFECTS
...
```

The sound effect initialization must be done only once. It can be done at any time, but it must be performed before playing the sfxs. The *SOUNDEFFECTS* label is found in the"*At3Files.asm*", and its label comes from the "label prefix" that you entered when exporting the sound effects.

Now, you can play the sound effect. In order to have a fully working example, I set up a little timer to play the first sound effect every 100 frames on channel 3:

```
...
    ;3 - Play one frame of the song.
    call PLY_AKG_PLAY

    ;3b - Plays a sound effect.
    ;Waits a bit before triggering a sound effect.
Counter: ld a,#0
    inc a
    cp #100     ;Waits for the 100th frame.
    jr nz,CounterEnd
    ;Plays a sound effect
    ld a,#1  ;Sound effect number (>=1)
    ld c,#2  ;channel (0-2)
    ld b,#0  ;Inverted volume (0-16)
    call PLY_AKG_PLAYSOUNDEFFECT

    xor a   ;Resets the counter.
CounterEnd:
    ld (Counter + 1),a

    ;4 - Loop!
    jr Sync
...
```

Remember that calling the *PLY_AKG_PLAYSOUNDEFFECT* method will only *program* the sound effect: it will only be heard when the PLY_AKG_PLAY is called, frame by frame.

Compile and test this code… This will work of course! For more info about how the sound effects work, check this page.

This is now the end of this tutorial. I hope it was useful to you, and that you will flawlessly integrate AT3 in your new productions!

Wrapper for Rasm

In this page, I'll explain how to generate a binary for the player and music, in case you don't want to use Rasm in your production. A better alternative is to use Disark!

First of all, have a small Z80 code, called "wrapper.asm", which we will compile with Rasm. It will embed the player and music:

```
  org #1000     ;This is the fixed address of the player/music
PlayerAndMusicBinary_Start
  include "resources/mysong.bin"
  include "resources/mysong_playerconfig.asm"

  include "playerAkg.asm"  ;It works for any other player.
PlayerAndMusicBinary_End
```

Note: if you're wondering what is that "player config" file, please check this page. This is optional, but don't be afraid to use this great feature!

Now, when compiling this, we'll ask Rasm to export the symbols as Z80 code.

```
rasm wrapper.asm -o wrapper -sc "%s equ %d"
```

This command will compile our "wrapper.asm", generating "**wrapper.bin**" (the binary), and a symbol file called "**wrapper.sym**". However, thanks to the magic "sc" command, **the latter has been generated as a Z80 file**. Indeed, the symbols have this structure:

```
PLAYERANDMUSICBINARY_START equ 4096
MYSONG_START equ 4096
...
PLY_AKG_INIT equ 5123
PLY_AKG_PLAY equ 5126
PLY_AKG_STOP equ 5129
...
PLAYERANDMUSICBINARY_END equ 6780
```

There are many labels present, but we don't care about them. Please note that Rasm generates upper-case labels.

So now, in your *real* code using your favorite assembler (Vasm, Winape, SJAsmPlus, etc.), simply do this:

```
  ;Loads the symbols for the music and player.
  include "wrapper.sym"

  ;Includes the music and player binary.
  org PLAYERANDMUSICBINARY_START
  include "wrapper.bin"

  org #4000
  ;This is my super game.
...
  ;Initializes the song.
  ld hl,MYSONG_START
  xor a   ;Subsong 0.
  call PLY_AKG_INIT
...
  ;This is the main loop.
  call PLY_AKG_PLAY
```

That's it! The PLY_AKG_INIT/PLAY labels can be conveniently used because they are declared in "wrapper.sym". You don't have to compile "wrapper.asm" again, unless the music changed, or you want to change its compilation address (from #1000 to anything else).

Please don't hesitate to contact me if you have any trouble making this work.

Source conversion with Disark

> This page is a stand-alone explanation of what is explained here.

You don't use Rasm because you already have an assembler you like, but your assembler does not understand Rasm mnemonics and macros. One possibility is to convert the player and music sources manually… But they are very complicated, as many macros/conditions are used. So are you stuck in including a non-relocatable binary of the player and music?

Well, no! A new command line tool has appeared: **Disark**. You can check its web-page to download it.

Disark is a disassembler, but can be used to **convert sources**. You can convert **any** source into your assembler, such as:

- SDCC / SDASZ80
- Winape / Maxam
- Pasmo
- Vasm
- Orgams
- Or use the default profile, which should work for many assemblers.
- If you want something specific, simply ask!

Once you have chosen your player and the music is ready, you only need to make **the conversion once**.

This is a 2-step process:

Assemble the sources with Rasm

- Download the Rasm for Windows, Linux and MacOsX here.
- Create a small source to include the player and the music. For example:

```
;No need to put a ORG or anything.

;Declare the hardware (check the player source for the flag to declare.
;CPC'ers don't need to add anything, CPC is default).
PLY_AKG_HARDWARE_MSX = 1   ;MSX is used here, for example.

;If sound effects, declares the SFX flag. Once again,
;check the player source to know what flag to declare.
PLY_AKG_MANAGE_SOUND_EFFECTS = 1     ;Remove the line if no SFX!

include "PlayerAkg.asm"        ;This is the AKG player.
include "MySuperSong.asm"      ;This is the music.
include "SoundEffects.asm"     ;SFX, if you have some.
```

Note (1): if you use the Player Configuration feature, you must include the configuration file(s) ***before*** *the player.*

Note (2): if you use the ROM players, the ROM flag/buffer location must be included too (check the player once again to know the flags).

- Assemble this source with Rasm like this:

```
rasm PlayerAndMusic.asm -o PlayerAndMusic -s -sl -sq
```

PlayerAndMusic.asm is the source above. The "*-o PlayerAndMusic*" indicates the base filename for the generated files (*PlayerAndMusic.bin* and *PlayerAndMusic.sym* (the symbol file)).

The *"-s -sl -sq"* options are **very important**, they ask to generate a *.sym* (symbol file) that Disark can use to recreate a faithful source.

Generate a source for your assembler, from the binary

Let the magic happen. Use Disark (in the *tools* folder) to convert the binary+symbol file into a new source file your assembler will like!

```
Disark.exe PlayerAndMusic.bin Final.asm --symbolFile PlayerAndMusic.sym --
sourceProfile sdcc
```

That's it! A *Final.asm* source has been generated for SDCC. You can include it directly in your SDCC sources, it compiles and is **relocatable**!

Of course the "source profile" parameter accept other values:

- winape, maxam, pasmo, vasm, sdcc, orgams.

You can also omit the *sourceProfile*, in which case Disark produces a source with no specificity at all, which is fine for many many assemblers.

Notes

- SDCC external labels are generated for the init/stop/play labels (if you use SDCC profile of course).
- One thing you should not do is mess with the generated source: for example, if you want to split the music and player into two sources, do not split the "disarked" source file: one part may reference the other (This is only a re-generated source after all). If you want to do that, do it right from the start, using two sources instead of one (one source will have the player, the other the music, and you will Disark them separately).
- Disark is a generic-purpose disassembler/source converter, so check out its web-page if you want to have your sources to be *Disarkable* (this is a new word I have just invented).

Using CPCtelera with AT3

`Note 1: this is the french version of this great article written by` **`Arnaud`**`, the author of many excellent games published mostly on the RetroDev competition. The` **`english`** `version is hosted on our good friends' website 64 Nops! Thanks again to Arnaud for his contribution.`

`Note 2: this tutorial was made for AT2, and the Lightweight (LW) player. This player is obsolete and no more included to AT3. However, you can apply it to all the other players, like AKG, AKM and AKY. Simply change the label name! An example is shown at the bottom. I also let the reference to "Arkos Tracker 2" in the text, don't feel threatened by that :).`

`Note 3: It is my intention to update this tutorial, to ease the use of CPCTelera with AT3. Please let me know if you are interested!`

L'objectif de ce tutoriel est d'utiliser des musiques et sons créés par Arkos Tracker 2 avec l'environnement de développement **CPCTelera**.

Beaucoup de jeux utilisent cet environnement, par exemple le gagnant du CPCRetroDev 2020 "The abduction of Oscar Z" (Dreamin`Bits), "Sorcerers" seconde place du CPCRetroDev 2020 (SalvaKantero) ou encore Space Moves (RetroBytes).
L'essentiel des participants au CPCRetrodev utilisent d'ailleurs ce Framework.

CPCTelera est un environnement de développement sous Linux ou Windows avec Cygwin et qui permet de programmer en C et en assembleur pour l'Amstrad CPC.
Il inclut des librairies écrites en assembleur permettant de gérer : les graphismes, le clavier, le firmware, la vidéo, la mémoire et l'audio. Pour plus de détails : http://lronaldo.github.io/cpctelera/

Actuellement CPCTelera ne gère que le player d'Arkos Tracker 1 et l'objectif de ce tutoriel est de pouvoir exploiter les musiques et sons générés à partir d'Arkos Tracker 2.
Le code assembleur généré par Arkos Tracker 2 peut être directement utilisé, seul le fichier de liaison (*PlayerAkm_cbinding.s*) devra être modifié en cas de changement de player par exemple.

Un exemple fonctionnel du "player Minimalist", basé sur les fichiers musicaux d'Arkos Tracker 2 "*SoundEffects.aks*" et "*Targhan – A Harmless Grenade.aks*", est fourni.

Pour le compiler il faut simplement entrer la commande make.

L'archive *arkos2.zip* est un projet CPCTelera classique :

- *songs* : contient les fichiers musicaux ArkosTracker2
- *src* : contient toutes les sources décrites dans ce tutoriel
- *main.c* : le programme principal permettant de jouer la musique et les sons
- *sound.c* et *sound.h* : un exemple d'implémentation du player d'Arkos Tracker 2
- *arkos2/PlayerAkm_cbinding.s* : fichier assembleur qui permet de faire le lien entre les appels des fonctions du C vers les fonctions du player d'Arkos Tracker 2
- *arkos2/PlayerAkm.asm* : fichier assembleur généré (cf. suite du tutoriel), compatible CPCTelera/SDCC, contenant musique/sons ainsi que le player Minimalist
- *arkos2/ArkosPlayer2.h* : fichier d'en-tête C contenant la déclaration des fonctions du player d'Arkos Tracker 2 pouvant être utilisées par le compilateur C
- *optional/setInterruptHandler.s* et *setInterruptHandler.h* : gestionnaire d'interruption (facultatif) utilisant les registres alternatifs

Génération et création des fichiers compatibles avec CPCTelera

Les étapes suivantes vous permettront d'utiliser vos propres musiques et sons dans CPCTelera.

- La première chose à faire est de suivre ce tutoriel (jusqu'au paragraphe "Regenerating sources" inclus) afin de générer le fichier assembleur au format "SDCC Z80".
  A l'issue de cette étape renommer le fichier généré en *PlayerAkm.asm* et le copier dans le répertoire *arkos2*.
- Deuxième étape : rendre accessibles les musiques et sons au compilateur C

Dans *sound.c* modifier le nom des ressources en fonction du nom de vos propres productions (leur nom est dans le fichier assembleur *PlayerAkm.asm*)

```
/** Resources */
extern void* AHARMLESSGRENADE_START;
extern void* SOUNDEFFECTS_SOUNDEFFECTS;
```

A la fin de cette étape tout est prêt pour être utilisé par CPCTelera.

> Remarque :
> Si vous voulez utiliser un autre player d'Arkos Tracker 2 (ex : le player générique AKG au lieu de AKM) il faudra modifier le fichier *PlayerAkm_cbinding.s* pour changer le nom des fonctions.

ex:

```
 _PLAYER_ARKOS_INITSOUNDEFFECTS::
     jp PLY_AKM_INITSOUNDEFFECTS
```

deviendra

```
_PLAYER_ARKOS_INITSOUNDEFFECTS::
    jp PLY_AKG_INITSOUNDEFFECTS
```

Mise en oeuvre des éléments créés

La première chose à faire est d'initialiser la musique et les sons avant de les utiliser (exemple du fichier *main.c*) :

```
// Main loop
void main(void)
{
    InitSound();
    …
}
```

Ensuite il faudra appeler la fonction *PlaySound* (dans l'exemple tous les 1/50ème de seconde) pour entendre le son produit.
Dans ce cas, l'appel à cette fonction peut être, soit dans une interruption :

```
void sInterruptHandler(void)
{
    static u8 sInterrupt = 0;
    // Play sound at 1/50th
    if (++sInterrupt == 6)
    {
        PlaySound();
        cpct_scanKeyboard_if();

        sInterrupt = 0;
    }
}

void main(void)
{
    // Init stuffs
    ...

    // Play first song
    PlayMusic(0);

    // Play sound on interrupt
    cpct_setInterruptHandler(sInterruptHandler);

    // Loop forever
    while (1)
    {
        if (cpct_isKeyPressed(Key_1))
        {
            // Play sound 1 on Channel A with sound Max
            PlaySFX(1, CHANNEL_A, MAX_VOL);
        }
        ...
    }
}
```

ou directement dans la boucle principale en utilisant une temporisation à l'aide de *cpct_waitVSYNC* :

```
void main(void)
{
    // Init stuffs
    ...
    // Loop forever
    while (1)
    {
        if (cpct_isKeyPressed(Key_1))
        {
            // Play sound 1 on Channel A with sound Max
            PlaySFX(1, CHANNEL_A, MAX_VOL);
        }
        ...
        PlaySound();

        // Wait for next 1/50th second
        cpct_waitVSYNC();
    }
}
```

Attention : il faut prévoir le cas où le son est joué sous interruption et que votre code utilise les registres alternatifs (la plupart des fonctions de CPCTelera ne les utilisent pas, mais vérifiez dans la doc).
Dans ce cas il faudra utiliser une version modifiée de *cpct_setInterruptHandler* de CPCTelera afin de sauvegarder et restaurer ces registres.
La fonction *asm_setInterruptHandler* fournie dans l'exemple permet de le faire en sauvegardant et restaurant tous les registres.

CPC Basic integration

`This example has been superseded by the interruption player (one call and the music plays in the background!), so you should probably use it instead.`

The players can all be called from assembler or C. However, some of us are still developing games on our good old Basic! Here is a little walkthrough on how to do that.

As a prerequisite, please download Rasm.

The example uses the AKG player, but it will work for all the other ones. Contrary to assembler, Basic requires a few securities, else it will crash, so we need to create a small assembler code to "secure" the calls to the players.

A few steps:

In the same folder, we will create our "secure" asm code. Either copy/paste the *sources/playerAkg/testers/Basic_CPC.asm* program where your music is, or use the code below. Save it as *Basic_CPC.asm*.

Export your music (as source), let's called it *MyMusic.asm*. **Warning**, make sure that the option "encode to address", at the bottom of each export window, is **unchecked** (we don't need to add an ORG to the music source as we directly include it to our own).

Copy the *PlayerAkg.asm* player in the same folder. It is located in the *sources/playerAkg* folder of the AT3 package.

```
;Basic_CPC.asm
    org #4000

    jp Init
    jp Play
    jp StopMusic

Init:
    ld hl,Music     ;Initializes the music.
    ld a,0          ;The subsong number (0 is the first one).
    call Player + 0
    ret

Play:
   di
   ex af,af'
   exx
        push af  ;Saves a few registers the system needs.
        push bc
        push ix
        push iy
        call Player + 3   ;Plays the music.
        pop iy
        pop ix
        pop bc
        pop af
    ex af,af'
    exx

    ei
    ret

StopMusic:
    di
    ex af,af'
    exx
        push af
        push bc
        push ix
        push iy
        call Player + 6      ;Stops the music.
        pop iy
        pop ix
        pop bc
        pop af
    ex af,af'
    exx
    ei
    ret

Player:
        include "PlayerAkg.asm"         ;The player. Loads the one you
want (AKG, AKM, Lightweight, AKY).

Music:
        include "MyMusic.asm"    ;The music.
```

Don’t be scared if you don’t understand assembler! Let’s compile this into binary:

```
rasm Basic_CPC.asm -o playmus
```

This will create a *playmus.bin* file, which contains the “secure” calls to the player, the player itself and the music.

Now load this file into Basic. There are many possibilities to do so:

- If you’re using Winape, you can inject it directly in memory (in the debugger, right click on the memory and load the file in #4000).
- Use ManageDsk by Demoniak to create a DSK and adds the file in it (as a binary, in #4000).
- For automated generation, some better tools exist, I let you search for them.

Load the file in Basic (unless you’ve injected it):

```
memory &3fff
load"playmus.bin",&4000
```

Now let’s play!

```
10 call &4000   'initialize the music.
20 call &bd19:call &4003   'plays one frame of the music, at 50hz.
30 if inkey(47)<>0 then 20  'loops as long as space is not pressed.
40 call &4006   'stops the music.
```

That’s it!

I chose the &4000 address, but you can change it (change the ORG in the source, as well as all the CALLs in the Basic example, and of course the LOAD address). Don’t forget to set the *memory* command one byte before, so that the Basic program doesn’t overwrite the binary (*memory &1234* if you load the music in &1235 for example). The address should be from &1000 to &9000, don’t go too high! Basic will prevent you from doing so anyway.

Finally, it is possible to use interruptions in Basic to play the music while doing something else. If you need such facilities, please let me know and I’ll create another example.

Using interruption player with CPC Basic

You want to play music and maybe sound effects with AT3 in your BASIC (CPC) production. You don't want to compile anything or go into complicated details because that bothers you. We got you covered! AT3 provides a small player wrapper that can be used easily on any Basic production.

I called it an "interruption" player because it uses system interruptions. Meaning that a simple CALL will play the song in the background, allowing you to run your production in the foreground, not having to bother with the music again! Plus, the included sound effect player will turn your program into an Hollywood production. Nothing less.

If you don't want to bother and only want to have fun:

- Open the "BasicInterruptions_CPC.dsk" in the "players\playerAkg\sources\tester" folder.
- Run "example.bas"
- Have fun!

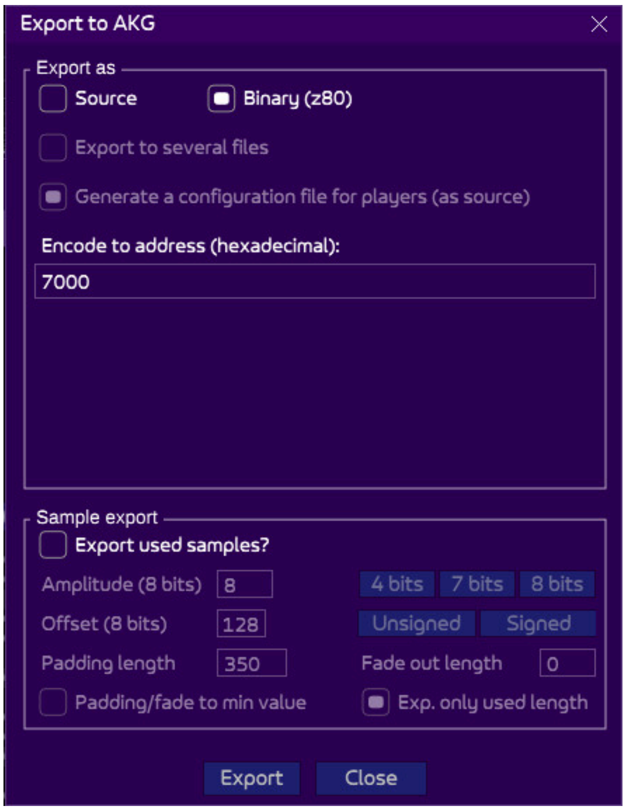
Now, if you want to use your own song and sound effect:

Exporting the song

The first step is to export the song. First, load your song. In my example, I use "A Harmless Grenade", which is in the song package of AT3. The obvious way is to use the File > Export > Export as generic (AKG) option.

> Note: AKG? Yes, the BASIC player is actually only a wrapper around the AKG (i.e. generic) player. So whenever you want to use the BASIC player, always use the AKG!

Since you don't want to compile anything (scary!), use the following options:


Export to AKG

Export as

Source [✓ Binary (z80)]

Export to several files

[✓ Generate a configuration file for players (as source)]

Encode to address (hexadecimal):

7000

Sample export

Export used samples?

Amplitude (8 bits) 8

Offset (8 bits) 128

Padding length 350

Fade out length 0

Padding/fade to min value

[✓ Exp. only used length]

Things to look after:

- Tick "export as… binary file (z80)"
- In "Encode to address", enter the address you want the file to be loaded in Basic, "7000", which is high enough in memory to let you have a big Basic program, but not too high because we will load the sfxs and player after, and they need memory too.
- Don't tick "export used samples", as the player does not support them, especially in Basic!

By clicking on OK, you can save the binary. Let's call it "music.akg". Make sure the music is not larger than #2000, else the sfxs are going to overwrite it.

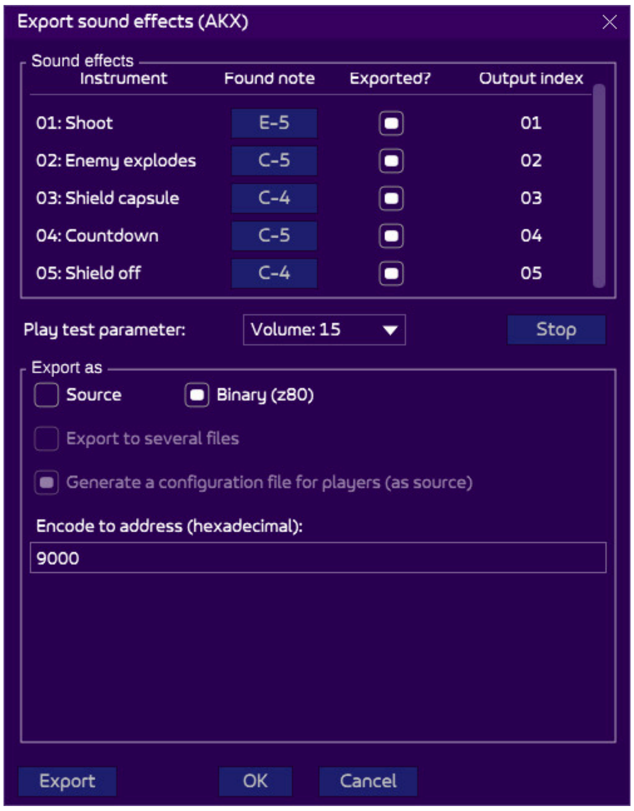
In the future I'm sure you'll be bored to do this step manually, so I advise you to use the command line tool (in the "tools" folder), which is much faster.

Export the sound effects

This step is not mandatory, but… what is a game without sound effects?

Let's open the "SoundEffects" song, still in the AT3 package, or load your own sound effects song.

Then click on File > Export > Export sound effects (AKX).


Export sound effects (AKX)

Sound effects

Instrument	Found note	Exported?	Output index
01: Shoot	E-5	✓	01
02: Enemy explodes	C-5	✓	02
03: Shield capsule	C-4	✓	03
04: Countdown	C-5	✓	04
05: Shield off	C-4	✓	05

Play test parameter: Volume: 15

Export as
Source [✓ Binary (z80)]
Export to several files
Generate a configuration file for players (as source)
Encode to address (hexadecimal): 9000

Use the following values:

- Make sure all the sound effects are exported (All "Exported?" ticked).
- "Export as… Binary file (z80)" must be ticked.
- In the "encode to address", write "9000", so that the sound effects are put just after the music, giving it about 8kb, which should be enough.

Click on Export and save the file as "sfx.akx".

Once again, this step can be automatized using the command line tool. Much faster!

Import in a DSK

Now we have the music and the sound effects. We need to put them in a DSK where you can then put a player and call it. As it sounds bothersome, we'll use a shortcut: there is already a DSK with everything in it. All you have to do is either overwrite the song and sfx on it, or simply use the current DSK and have fun with it.

Make a copy of the "BasicInterruptions_CPC.dsk" in the "players\playerAkg\sources\tester" folder.

Are presented here three ways to import your files:

ManageDsk

You can use the venerable ManageDsk from Demoniak, Windows only. Drag'n'drop the music and sound effects files, making sure that:

- Choose BINARY when importing.
- Set the right start address: 0x7000 for the music, 0x9000 for the sfx (don't set the Exec).

Cpcfs

This tool by Ramlaid is Windows only, command line. You can import the files by typing this:

```
cpcfs BasicInterruptions_CPC.dsk p music.akg,0x7000 -b -e
cpcfs BasicInterruptions_CPC.dsk p sfx.akx,0x9000 -b -e
```

dsk

dsk by Sid/Impact is a Linux/Mac/Windows command line tool doing the same thing as cpcFs above. Works very well!

How to use

So far, we have the song and the sound effects file. Open the "BasicInterruptions_CPC.dsk" in the "players\playerAkg\sources\tester" folder. The player is already compiled in 0x9500.

Simply loads the files:

```
10 memory &6fff
20 music=&7000
30 sfx=&9000
40 player=&9500
50 load"music.akg",music
60 load"sfx.akx",sfx
70 load"player.bin",player
```

So far, nothing fancy. If you don't use sound effects, simply don't load the sfx file.

> Note: Make sure your music doesn’t go to overwrite the sfx file, and that the sfx file doesn’t grow over the player! The player is high enough in memory to be “safe”, both for Basic and the system. You shouldn’t have to recompile it, but you may if you want to relocate it elsewhere. Yes, it means using Rasm.

Then you simply call the player to **play your song**! This should be done once, but if you stop the music and want to start again, this is the command to execute. Also if you want to restart a song, or play a different subsong.

```
90 call player,music,0
```

Where “0” is the subsong number (0 being the default song).

Then, if you use sound effects, now is a good time to **initialize** them too. This should be done **only once**, at **any time**, **before** using the sound effects. Of course, if you don’t use sound effects, skip this step.

```
100 call player+6,sfx
```

To **stop the song**, only one simple command:

```
call player+3
```

Now, let’s play some sound effects!

> **Note that the sound effects will only be heard if the music is playing!** Once the music is stopped, **the sound effects will not be played**! If you want sound effects but no music, simply play an empty music. See below.

To **play a sound effect**, one simple command:

```
call player+9, <sound effect number>, <channel>, <inverted volume>
```

The “sound effect number” starts at 1 and matches the exported sound effect in the screenshot at the top of the page.

The “channel” is 0 (left), 1 (center) or 2 (right).

The “inverted volume” is the volume, from 0 (full volume) to 16 (mute). Example:

```
call player+9,2,1,0
```

This will player the sfx 2 (“enemy explodes”), in the center channel, at full volume.

One last command. It will **stop a sound effect** from a channel:

```
call player+12,"channel"
```

“channel” is as above, from 0 to 2. This will stop **any** sound effect that is playing **on the channel**.

Muting the music

You might want to mute the music, or simply have no music during your production.

The latter is simple: simply generate an empty song, and start it just like shown above.

To mute the music, simply add another Subsong in your song (the main song being Subsong 0 by default) with nothing inside. Then whenever you want to mute the music, play the Subsong 1 (the muted Subsong). That’s it! The only drawback is that if you want to unmute the music (and thus play Subsong 0), it will start at the beginning.

Wrapping up

That’s it! This should be enough to get you started. Have fun!

Using interruption player on ZX Spectrum

The following code will allow you to play a music under interruption, on ZX Spectrum (without using the firmware).

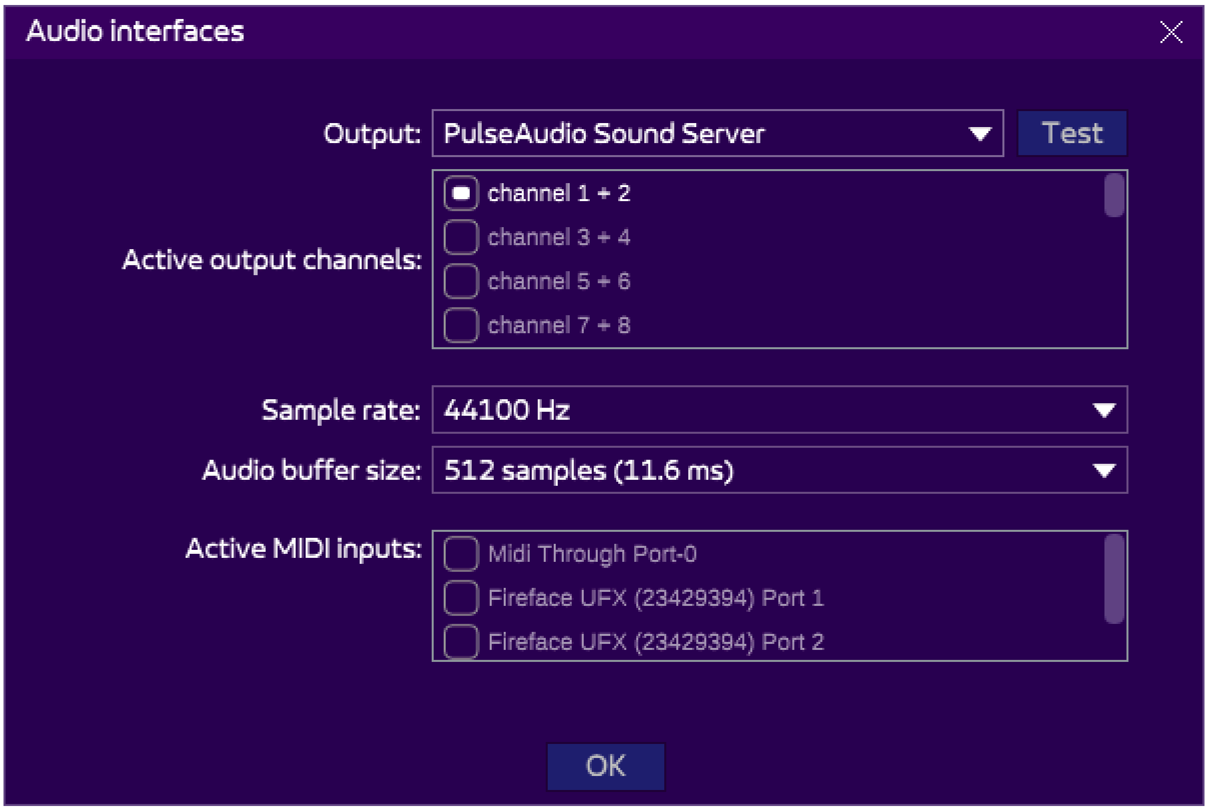
This code was kindly provided by **Gusman**. A big thanks to him!

```
PrepareMusic:
    ld hl, MUSIC_ADDRESS ;load your song address
    xor a ;subsong 0
    call PLY_AKG_INIT ;call initialization routine
    ret

PlayMusic:
    call Interrupt_Setup ;This function is totally unnecessary, just for
readability of the code
    ret

StopMusic:
    di ;disable interrupts
    call PLY_AKG_STOP ;stop the player

    IM 1 ;revert interrupt mode to mode 1
    ei ;enable interrupts

    ret

Player:
    ;your player+music code
    #include "music.z80asm"

JP_ADDRESS EQU #FDFD ;JP address

;IM 2 mode must be used on the spectrum to have a custom interrupt
handler.
;The spectrum architecture makes impossible to know what will be on the
bus when an interrupt is executed
```

```
;so the handler must be in an address where it's high and low bytes are
equal.
;To be able to store your interrupt handler anywhere three bytes at an
address
;with that characteristic (FDFD in this case) are reserved and written
with "jp Interrupt_Handler"
Interrupt_Handler:

    push af ;store all registers
    push bc
    push de
    push hl
    push ix
    push iy
    exx
    ex af, af'
    push af
    push bc
    push de
    push hl

    ;play music
    call PLY_AKG_PLAY
    pop hl

    ;restore all registers
    pop de
    pop bc
    pop af
    ex af, af'
    exx
    pop iy
    pop ix
    pop hl
    pop de
    pop bc
    pop af
    ;reenable interrupts
    ei
    ;return from interrupt handler
    ret

.align 256
Interrupt_Table:
    ;interrupt table must be aligned at page boundary
    ;with 256 + 1 bytes, all with the same value
    .defs 257

Interrupt_Setup:
    di ;Disable interrupts
    ld de, Interrupt_Table ;load interrupt table address
    ld hl, JP_ADDRESS ;load "JP" address
    ld a, d
    ld i, a ;load I with the interrupt table high byte
    ld a, l ;load a with lower byte of JP address (indifferent to use H or
L, both must be equal)

Interrupt_Table_Loop:
    ld (de), a ;fill the table
    inc e
    jr nz, Interrupt_Table_Loop
    inc d ;write the 257th byte of the table
    ld (de), a
    ld (hl), #C3 ;write JP
    inc l
    ld (hl), low(Interrupt_Handler)

    ;write interrupt handler address
    inc l
    ld (hl), high(Interrupt_Handler)

    im 2 ;set interrupt mode to 2
    ei ;enable interrupts

    ret
```

Audio/midi setup

By selection the File > Setup > Audio/MIDI interfaces, a dialog opens:


Audio interfaces

Output: PulseAudio Sound Server

Test

Active output channels:
- [✓] channel 1 + 2
- [ ] channel 3 + 4
- [ ] channel 5 + 6
- [ ] channel 7 + 8

Sample rate: 44100 Hz

Audio buffer size: 512 samples (11.6 ms)

Active MIDI inputs:
- [ ] Midi Through Port-0
- [ ] Fireface UFX (23429394) Port 1
- [ ] Fireface UFX (23429394) Port 2

OK

At the top is the **output**, which varies according to your setup. On Windows, it is advised to use ASIO drivers (if your soundcard allows it, or ASIO4All as a fallback) if you want to lower the latency as much as possible.

The **active output channels** indicates where the sound is output. This is the output you probably plugged your speakers or headphones, or where your laptop has its speakers rigged to.

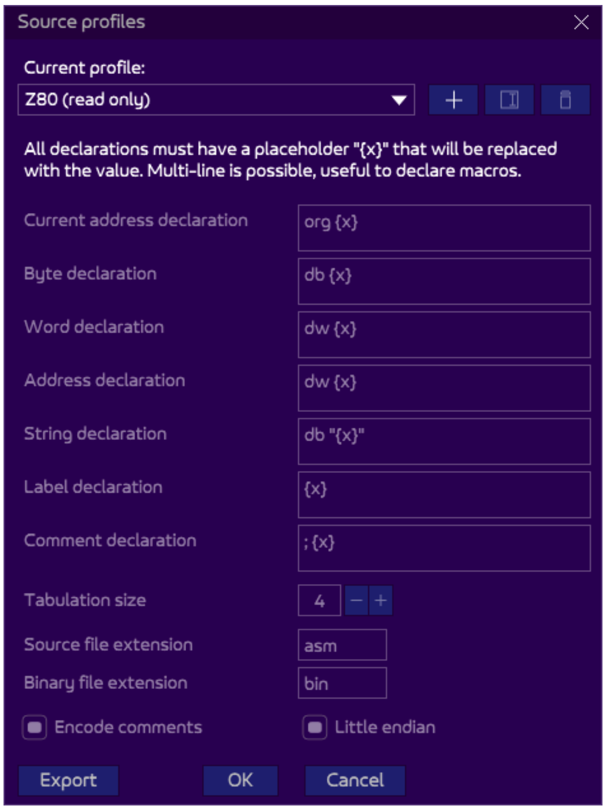
A **sample rate** of 44100 Hz is enough, more is overkill.

The **audio buffer size** has a direct relationship with the latency (a high latency provokes a noticeable duration between you pressing a keyboard note and actually hearing it). The lower the better obviously (less than 20 ms is advised), but the smaller the buffer, the more overhead it produces on your CPU. If you hear crackles and pops when playing a song, raise the buffer size. The quality of your sound card greatly influences the latency.

At the bottom are the **MIDI inputs**, for you to play notes with your MIDI keyboard. If you had used a MIDI keyboard previously, it should be selected as active. If adding a new MIDI keyboard, make sure it is selected here.

Source profiles

Access this via File > Setup > Source profiles:


Source profiles

Current profile: Z80 (read only)

All declarations must have a placeholder "{x}" that will be replaced with the value. Multi-line is possible, useful to declare macros.

Current address declaration	org {x}
Byte declaration	db {x}
Word declaration	dw {x}
Address declaration	dw {x}
String declaration	db "{x}"
Label declaration	{x}
Comment declaration	; {x}
Tabulation size	4
Source file extension	asm
Binary file extension	bin

Encode comments | Little endian

Source profiles is a powerful feature when exporting your song as source. Indeed, you may want to target a specific CPU, and more importantly, a specific assembler. You want your assembler to be able to understand the exported source. A few factory profiles are already present, but you may want to create specific ones.

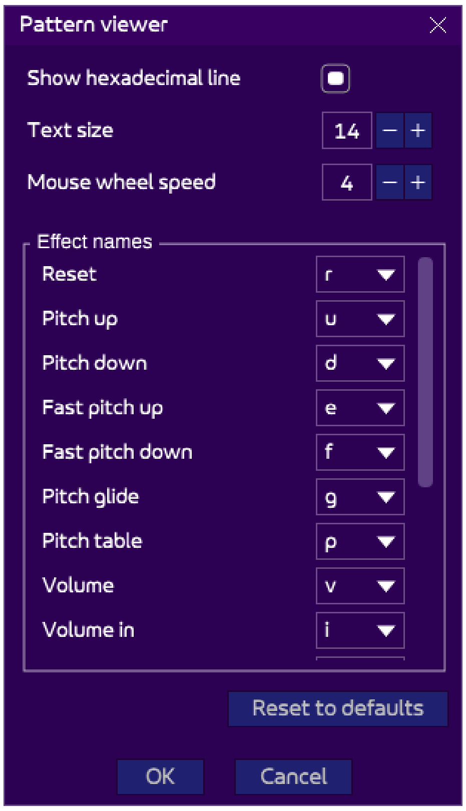
Factory profiles are all marked “read only”, as the greyed-out items show. Press the Plus icon at the top-right to duplicate the currently selected one. The new profile is ready for modification. You can rename and delete it via the icons at the top-right.

Each item must have a “{x}” which will be substituted with the value to encode. This enables the most complex syntax and as multi-line is possible, you can even encode macros if you wanted to.

An export button at the bottom-left exists to use the profile into the command line tools.

Pattern viewer setup

Opens the Pattern Viewer (PV) setup via the File > Setup > Pattern viewer menu:


Pattern viewer

Show hexadecimal line

Text size 14

Mouse wheel speed 4

Effect names

Reset	r
Pitch up	u
Pitch down	d
Fast pitch up	e
Fast pitch down	f
Pitch glide	g
Pitch table	p
Volume	v
Volume in	i

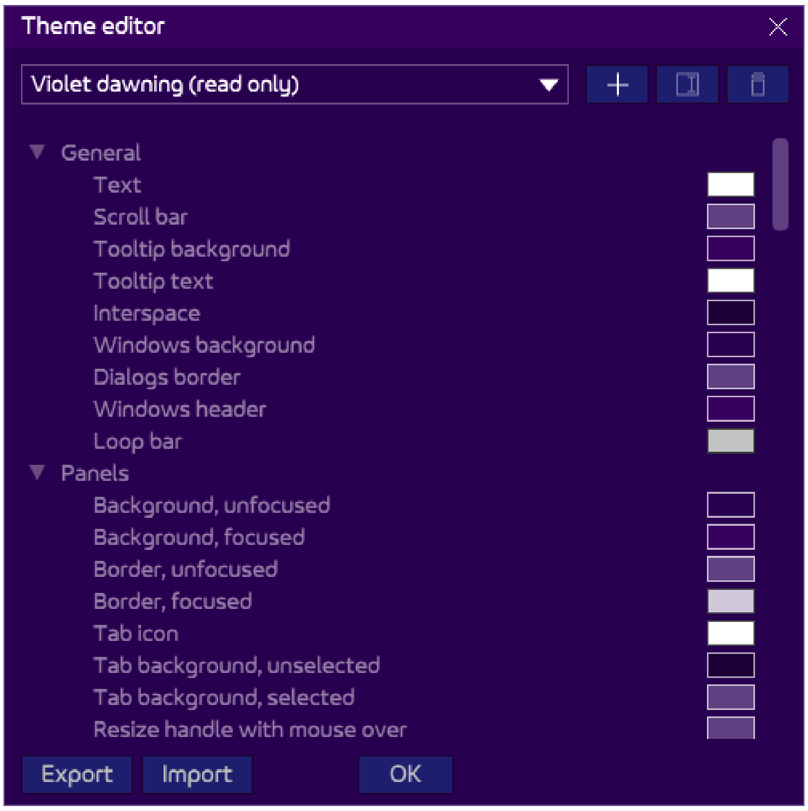
The top options are pretty self-explanatory.

However, the effect names may not be, which you can see the default values here. These are directly seen in the PV when effects are present. In most (all?) trackers, an effect shows as a letter, and cannot be changed. This is possible in AT3!

For example, if you want the Reset effect to be seen by another letter or digit than “r”, simply select a new one in the drop-down. Duplicates are written in red. Note that you obviously have to type the right letter too in the PV!

Theme editor

Open the theme editor via File > Setup > Theme editor:



- General
  - Text
  - Scroll bar
  - Tooltip background
  - Tooltip text
  - Interspace
  - Windows background
  - Dialogs border
  - Windows header
  - Loop bar
- Panels
  - Background, unfocused
  - Background, focused
  - Border, unfocused
  - Border, focused
  - Tab icon
  - Tab background, unselected
  - Tab background, selected
  - Resize handle with mouse over

A great feature of AT3 is the ability to change all the colors used in the user interface. The drop-down at the top allows you to select a pre-defined theme, or a custom one you have done by yourself or imported.

If you're not satisfied with what is available, you can start a new one from an existing one. Note that factory themes are all marked as "read only", making modifying impossible, as show the greyed-out items in the tree below.

Start a new theme by selecting one as a starting point and press the Plus icon at the top-right. It is automatically selected, and is free for modification. It can be renamed and deleted at will, still with the icons at the top-right.

When clicking on a color, a swatch dialog opens and you can choose any color you want to replace it. The change is real-time.

Pressing OK will save your modifications. If you are satisfied with your theme, you may want to export it (see the button at the bottom-left) to save it preciously, or even send it to the AT team for an inclusion in the next release! Fame at last!

FAQ

Is it an open-source project?

Yes.

Can I use the players in my production?

Of course! The players are MIT-licensed. Basically, you can use and modify them at will, in any production, free or sold, open or closed source.

One nice (but non-mandatory) thing you can do would be to put a credit somewhere in your production about Arkos Tracker 3, and we'll be best friends forever.

What assembler to use to compile the Z80 sources?

Rasm, but don't worry, thanks to Disark, you can convert the sources to any assembler! Please check this page for more information.

I can see Z80 sources. What about Atari ST, Vectrex and Oric players?

The 68000 (Atari ST) AKY player has been brilliantly done by ggn. The latest version can be found on ggn's repository here.

6502 AKY player (Oric/Apple 2) is now done, thanks to Arnaud Cocquière.

Vectrex AKY player is coded by other people which *may* still work on it. There is even a Dragon version on the way!

I want to help and convert the Z80 sources into the XXX processor

… But the sources are so complicated! Indeed, with all the conditional assembling, the sources are hard to decipher. The solution is to clean the sources by using Disark, which will recreate the source, without all the macros and conditions.

Should I still use Arkos Tracker 1 and 2 instead of 3?

No. AT1 is an old stuff. AT2 is not supported anymore and cannot be modified easily. Enjoy AT3!

Is there a sample player available to play the songs including samples?

A 3-channel MOD player (Amtrad CPC only) is included. A digidrum player is also available. For a PSG+one sample player, check here.

Will a half-track player be released (two PSG tracks, two digi channels)?

Such players are possible, I did it for the Landscape part of the DemoIzArt. I may do one under popular demand, which hasn't shown yet.

Will there be ST-SID one day?

Who knows, maybe? Depends on how much you ask for it. There is nothing that can prevent AT3 from including SID support. The problem lies on the hardware: SID players are quite tricky to do, and a player I would provide may not fit all the needs, resulting in a player that no one would use. Plus, to be honest, ST SID really sounds dull.

Troubleshooting

On Windows, a dialog opens about a missing DLL

This has been reported to happen on Windows 10: “vcruntime140.dll is missing”. Simply unzip this zip (for 64 bits), which contains 2 DLLs, and copy them both in `c:\windows\SysWOW64`. Thanks to Reset for this solution!

This link can also help you for, at least Windows 10, and probably 10. Thanks dthrone for this link!

On Windows, if I move the window on one corner, I expect it to be resized to a row or column

AT3 custom title-bar prevents the native behavior of your OS to take control of the window. Go to File > Setup > General, and tick “use native title-bar”. Thanks Cracky for notify me on this!

On Mac, the program doesn’t run, it is considered “damaged and can’t be opened”

This is “normal”, because the app is not signed (it would require a 99$/y Apple Developer account which I’m not ready to purchase). You must tell the system the app is all right.

There are a few ways of doing it, explained here, from a simple click to a command line trick.

The simplest way is:

- Right-click (or Control-click) on the app icon.
- Select **Open** from the context menu.
- A dialog will appear; click **Open** again. This will allow you to run the app without changing security settings.

However, depending on your OS and the level of security it has, you may need the most brutal (yet efficient) way:

Open the command line, locate “ArkosTracker3.app” in the unzipped package, and type:

```
xattr -cr ArkosTracker3.app
```

Then double-click on the app again using the Finder.

On Linux, the program crashes right from the start with a stack trace

You might be missing some dependencies. You can try:

```
ldd ./ArkosTracker3
```

to find the culprit, if you are Linux-savvy enough. If not:

On Mint 20.1, these dependencies might be missing (install one, then run the software, then install the next one if it doesn't work):

```
sudo apt-get -y install libx11-dev
sudo apt-get -y install libfreetype6-dev
sudo apt-get -y install g++
```

Arkos Tracker requires all the following dependencies. You might already have most of them, so it shouldn't be needed to install all of them! If in doubt, there shouldn't be any problem if you install them all.

```
sudo apt-get -y install libfreetype6-dev
sudo apt-get -y install libx11-dev
sudo apt-get -y install libxinerama-dev
sudo apt-get -y install libxrandr-dev
sudo apt-get -y install libxcursor-dev
sudo apt-get -y install mesa-common-dev
sudo apt-get -y install libasound2-dev
sudo apt-get -y install freeglut3-dev
sudo apt-get -y install libxcomposite-dev
sudo apt-get -y install g++
```

Another possibility is described by **kalantaj** in the forum (thanks!):

According to the stacktrace, such as:

…

```
/lib/x86_64-linux-gnu/libpthread.so.0(+0x153c0)
/lib/x86_64-linux-gnu/libGLX.so.0(+0xa679)
/lib/x86_64-linux-gnu/libGLX.so.0(glXChooseVisual+0x17)

...
```

Go to Synaptic ("sudo synaptic" in command line), enter the name of the library that is shown in the stacktrace and that is probably missing (libGLX for example) and install the missing packages.

On Linux, no file picker shows when I want to load/save a song or instrument!

Some Linux desktops don't have one (Manjaro or XFCE). Please install **kdialog** or **zenith**, and their file picker will be automatically used.

On Linux, loading/saving a file shows a custom file chooser. Can't I use the one from my OS, which I like?

Yes, it should be default, but some Linux desktops don't have one (Manjaro or XFCE for example). Please install **kdialog** or **zenith**, and their file picker will be automatically used.

On Linux, serial communication doesn't work, I'm sure the set-up is correct!

The serial port may be read-only. You can make it read-write using such command:.

```
sudo chmod 666 "port name"
```

The port is usually something like "dev/ttyUSB0", but you can get this list in the Serial setup.

I have a high latency on Windows, there is one second between pressing a key and hearing the sound!

This is related to your sound card and/or its drivers. Three possibilities:

- In AT3, go to File>Setup>Audio interfaces, and decrease the "Audio buffer size" as much as possible.
- If it doesn't work, try changing the audio device type at the top of the same window.
- Still not better? You will have to install a low-latency driver. A good sound card should allow ASIO (you should have seen it in the audio device type), but if not, install the ASIO4All drivers.

Don't worry, it's harmless. Once it is done, don't forget to select this driver still in the audio device type list.

Exporting in binary fails!

Please make sure your export address is not too high! Your music may be too big, so the embedded assembler (Rasm) failed to compile it, because you've gone past &FFFF. Z80 binary export can only target a 16 bits memory range!

The music doesn't sound the same on the hardware!

Make sure you have selected the correct PSG frequency in the Song Properties, as well as the right PSG (AY/YM)!

The player crashed/doesn't work properly

Before sending me the music for me to test (mandatory!), please check that:

- The **stack** is in a safe location.
- You used the **right player** with the right song. Using the AKG player with a non-AKG generated song will crash, obviously.
- To be on the safe side, make a test **without** the player configuration files.
- To be on the safe side, make a test with the **RAM** player, not the **ROM** player, if possible.
- If you were using Disark, try to make a test without, and only use Rasm and a non-relocatable binary of the player.
- Remove the use of **sound effects**, if you were using some.
- Did you use the right **player configuration files**? If you don't, the player will crash indeed. When using the player configuration feature, the player is tailored to a specific song(s), so using the same player with another song will surely crash!
- According to your system, **do the interruptions need to be disabled when calling the player** (shouldn't be necessary, but do it just to test)?
- Does the system of your OS require some **registers to be saved**? If yes, then do so (for example, on CPC, the **system (SYSTEM, I said)** interruption handler requires BC' and AF' to be saved).
- Please bear in mind that the players, most of the time, **modify ALL the registers**, including IX, IY, and the auxiliary ones! **Save them** if needed! **Save them all if using the player in an interruption handler!**

The app crashed!

This is unlikely. Really. However, logs may have been saved. Please copy and paste them to me. They should be found, depending on your platform, on these locations:

- On Windows: c:\Users\<username>\AppData\Roaming\ArkosTracker3\crashlog.txt
- On Linux: home/<username>/.config/ArkosTracker3/crashlog.txt
- On Mac: ~/Library/Logs/ArkosTracker3/crashlog.txt

Contact

A remark, a bug, a feature request? Any feedback, positive as well as negative, is **vital** for such project. Don't hesitate to contact me to provide some! Email me at contact at julien-nevo dot com.

You can also follow me on TwitX.

Want a real-time answer with me or some other helpful folks? Then join the Amstrad CPC Community Discord, it's very friendly! A special Arkos Tracker section has been opened just for you.

Credits

Arkos Tracker is coded and designed by **Julien Névo** a.k.a **Targhan/Arkos**.

Application logo by **Leïla R**.

Players

- all Z80 players by **Targhan/Arkos**, except:
    - FAP player **Hicks/Vanity** and **Gozeur/Contrast**.
- 68000 AKY player by **ggn**.
- Apple2/Oric 6502 AKY player by **Arnaud Cocquière**.
- Atari XE/XL 6502 AKY player by **Krzysztof Dudek**.

Huge thanks to them for their hard work!

Contributions

- **Mr. Earwig** for the Docker folder, to build the software from a container.

Third parties software

- Rasm by **Roudoudou/Flower Corp** – a Z80 assembler, used to assemble the generated Z80 sources into binaries.
- FAP player and cruncher by **Hicks/Vanity** and **Gozeur/Contrast**.
- LZH depack code by **Haruhiko Okumura** (1991) and **Kerwin F. Medina** (1996).
- C++ wrapper by **Arnaud Carré** (Leonard) – used to depack YM files.
- Serial by **William Woodall** and **John Harrison**, a cross-platform serial port communication library.
- SUZUKI PLAN by **Yoji Suzuki**, a Z80 emulator used for the players test units.
- Speaker emulation based on RBJ Biquad HighPass by **Vincent Falco**.

Special thanks

Thanks to **Grim/Arkos** for the AY volume measurements.

Thanks to **Benjamin Gérard** for the YM volume measurements.

Thanks to **Zik** for his insights, suggestions, and the nice debates about players and music software.

Thanks to **TotO** for his PlayCity and great CPC hardware.