The Tracker's guide to MSX music with MuSICA

I think that the MSX is an underdog in the western world of 8-bit computing. Pretty much unheard of in the USA, enjoying moderate success in Europe, it was still quite popular in Asia and South America. For those wanting to create music with the machines, English language resources have been pretty limited, so after some of my own experimentation with the machine I compiled this small guide.

The standards and the hardware

MSX is a standard rather than a single model, so many hardware manufacturers produced different models adhering to it. If you want to know all the details I guess you can check the Wikipedia article out, but sound-wise the standard included the AY-3-8910 PSG chip, in later standard revisions (MSX2, MSX2+) to be replaced with YM2149. It means that in terms of what it can do, it's pretty much compatible with the "higher-end" Spectrum models and the Atari ST (minus the CPU heavy tricks of the ST).

Additionally to the MSX standard, there are two optional standards that are built into some models and added through a cartridge port on others: MSX-MUSIC and MSX-AUDIO.

The MSX-MUSIC standard consists of an YM2413 chip (OPLL) and a ROM extension with routines to use it in, for example, BASIC programs. The OPLL is an OPL2-like FM chip that is quite limited in that it can only use one user-generated instrument at a time while the others are basic patches built into a ROM in the chip. It can be used in a nine-channel polyphonic mode, or three channels can be sacrificed to provide five channels of simple drum sounds along with six FM channels.

The MSX-AUDIO standard is similar to the MSX-MUSIC in that it uses a Yamaha FM chip, namely the Y8950. It's OPL1-compatible, but provides a hardware sample player and DAC.

Among the two optional sound standards, MSX-MUSIC enjoyed the highest popularity, with the YM2413 being a bit cheaper, I guess.

Some common MSX-MUSIC compatible cartridges are FM-PAC, FM PAK or FM STEREO PAK (Dutch clones of the FM-PAC). As for MSX-AUDIO, the only one I've found is the Philips Music Module which apparently contains a MIDI interface, too.

Konami SCC

Further adding to the confusion, Konami used a proprietary sound chip in many of their game cartridges that eventually became a de facto standard among composers. The chip allows for five channels of user-definable waveforms to be played, maybe not so much unlike the PC-Engine or the Game Boy wave channel. I don't really know a lot about this

hardware, but for a good example of the sounds it is able to produce, check out the game "Space Manbow".

MuSICA

The software I'm going to focus on in this guide is an MML editor/compiler that runs on the actual hardware or in an emulator, allowing you to compose for the built-in PSG, MSX-MUSIC and SCC at the same time. It's quite simple but still flexible enough to allow for some nice music. As this is the "Tracker's guide" I'll try to present the information in a way that makes sense to someone who's used to composing in a tracker or piano roll editor.

Channel definitions

When you first start the program, you're presented with a text editor screen with a few rows of keywords representing the different channels. I will call these keywords **channel definitions** from now on. FM1 through 9 represent the melodic FM channels, FMR represents the rhythm channels, PSG1 through 3 represent the PSG channels and SCC1 through 5 unsurprisingly represent the five SCC channels.

In these channel definitions you do the over-all arrangement of the song built from **sequence definitions** (I'll get back to these later. In short, they represent a sequence of notes and/or commands, similar to tracks in trackers). For example, a channel definition could look like this:

```
FM1=t, s, s1, s2, s3/2
```

In this case, when playing the song, it would first play back the notes/commands in the "t" sequence definition, then is would play s, then "s1" and then "s2". When it reaches "s3/2" it will play the "s3" sequence twice. If you ad "/n" (with n being a number. I will keep using n to represent numbers from now on) after a sequence in the channel definition, it will play it n times. This is very useful for compact repetition.

The channel definitions are very flexible in that the playing of the different channels is asynchronous. You can have variable length of the sequence definitions, and even you can even use different tempo for all the different channels if you want.

As far as I know, this is all there is to channel definitions.

Sequence definitions

This is where the music is happening. In a sequence definition, you can have any number of notes and commands in any order as far as I know. To create a sequence, just add another row in the document and write "name=" with name being what ever you want to call the sequence. Any name seems OK, as long as they're four characters or shorter.

Let's continue the example in "Channel definitions" and write a definition for the first sequence, "t":

```
t=t150
```

In this case, the sequence t contains a single command, **t**n (n being a number again). "**t**n" sets the tempo for the channel it is used in, so in this case the tempo for FM1 is set to 150. If you want all channels to have the same tempo, you can use this sequence definition in all the channel definitions like so:

```
FM1=t, s, s1, s2, s3/2
FM2=t
FM3=t
FM3=t
...
```

The tempo command is just one of many available, but I don't think you should worry too much about the different commands just yet. Just read along, and I'll explain some of the more common ones and at the end of the document I will provide a more comprehensive list of them.

Now, let's write a definition for the sequence "s", which is the next sequence in the FM1 channel definition. I usually use the second sequence in the channel definition to setup different channel parameters such as instrument patch, octave, default note length and volume.

```
s=@10o4l8v15
```

This is a sequence of four commands; @n (use instrument patch n), **o**n (set current octave to n) **l**n (set the default note length to 8^{th} notes. I will explain more on this later.) and **v**n (set channel volume to n). These four commands are executed instantly without pause as they are being read, so they won't delay the other patterns. Pretty straight forward so far, isn't it? If it isn't, take a rest and I think you will understand it eventually.

Let's get going with the next sequence then, "s1". This is the first sequence in which I put a number in the name. This has no technical relevance and as I said you can name them whatever you want, but for the sake of clarity I use unnumbered names for sequences that contain just commands, and I use numbered names for sequences that contain musical data. This works well for me, but you can use whatever naming convention suits you.

Ok, let's look at s1 then:

```
s1=cde.f16gfedc4r2
```

First off, I will tell you that this is a sequence of notes. Notes are named c, c#, d, d#, e, f, f#, g, g#, a, a#, b. There is also a special note, r, that behaves just like the note command but represents a rest. If you don't put a number after a note, it will use the default note

length (remember the l8 command?). The note length is expressed in fractions of a measure, so in "tracker terms", at speed 6, l32 means half a step, l16 means a single step, l8 means two steps, l4 means four steps, l2 means eight steps, and l1 means 16 steps.

This should make most of s1 pretty clear, but there's still one command that isn't covered: . (the dot). Those of you familiar with western notation will recognize it as a dotted note. In simple terms, adding a dot after a note of any length will increase the note length by 50% of the original note. Adding another dot will further increase the length further by 25% of the original note, and another will increase the length further by 12.5% of the original note. So, "c4.." for example, will produce a note of 4+2+1 steps (in tracker terms) or in more general musical terms, a fourth extended by an eighth and a sixteenth.

In the case of "s1" we use it to extend the first E by a 16^{th}, and then we the first F shorter so that the length of both adds up to a 4^{th}. If you have taken singing or piano lessons, you might recognize this as "Do re mi fa so la ti do do" with a little syncopation. If not, with any musical background it will still make sense when you listen to it!

Ok, all clear? Experiment some and compare with a music program you're familiar with if it still isn't. Let's continue with s2!

```
s2=cd#ff#ga#>(c16<a#16>)c4r2<
```

Here you can see a few new things, but if you understood s1 you will recognize the notes as part of a simple blues scale. The first new command is >. With this you simply increase the octave number by one. Accordingly, you can also decrease the octave number using <. I use a final < at the end of the sequence to make sure I end up on the same octave I started on when I start the next sequence.

The second new thing is the sequence of notes encapsulated by "( )". This sequence of notes is played as a glissando. If you think of it as a guitar, the first note of the encapsulated sequence is plucked, and the following notes up to and *including the immediate note after the end parenthesis* are played just by moving your finger over the frets without plucking again. In this case, it means that "(c16<a#16>)c4" is played as a continuous sound starting at c16 and ending after c4. If it doesn't make sense, listen and experiment!

Finally, in the sequence "s3" we will make it play a simple octave bass:

```
s3=o3l16@23dd>dd<dd>dd<dd>dd<dd>cd<
```

This sequence begins with the commands we used in "s". As I said, any notes and commands can be used in a sequence definition. In this case I want to change patch, octave and default length, and then I put in a sequence of notes and octave shifts.

Now we have made a channel definition for FM1 and definitions for all the sequences used in it:

```
FM1=t, s, s1, s2, s3/2
...
SCC5=

t=t150
s=@10o4l8v15

s1=cde.f16gfedc4r2
s2=cd#ff#ga#>(c16<a#16>)c4r2<
s3=o3l16@23dd>dd<dd>dd<dd>dd<dd>cd<
```

We can try to compile and play it all using F5 on the keyboard. While it's playing, you can fast-forward using F4 and you can stop using F3

Also, let's save it! Press "Escape", press D for disk and press 3 to save music and you will be prompted to enter a file name. Note that the file name HAS TO BE eight characters. That means that if it's shorter you have to pad with spaces. Simply name it "test " (with four spaces afterwards). You can now try loading it too, just to get familiarized. Again, the file name has to be eight characters! When you're done, you can return to the editor screen by pressing Escape.

Now we're going to try to define the FMR (rhythm) channel, which is quite different from the others. First of all, start a new song by pressing Escape > N > Y. Now, simply define FMR as:

```
FMR=t, r, r1/8
```

This should all be familiar so far.

Let "t" be the tempo, whatever you want it to be:

```
t=t150
```

Let's just put the volume in "r", because the other commands don't work anymore.

```
r=v15
```

Finally, the really new kind of sequence is "r1":

```
r1=bh8h8sh8h8
```

This is how rhythm notation works: first you specify what drums are being hit at once, and then you specify the length to the next drum hit. So in this case, we play bass+hat,

rest an eighth note, hat, rest an eighth note, snare+hat, rest an eight note and finally hat, rest an eight note. The five different drum sounds are b (bass drum), h (hi-hat), s (snare) c (cymbal) and m (tom). You can change the volume of the individual drum sounds by writing for example “vb12”. This will set the volume of the bass drum to 12.

Try playing the sequence. Then, using what you’ve learned, add the Billie Jean bass line on the FM1 channel!

Note that you can’t use FM7-9 when you use the rhythm channel. The operators of these three channels are used together to produce the rhythm sounds.

The voice editor

In the main editor, press Escape > V to enter the voice editor. Here you can edit all the FM patches and create new ones. You simply move the cursor around using the arrow keys, and you increase/decrease the values using Space and N respectively. Note that you can’t change patches that are marked with “*”, because these are the built-in patches of the OPLL chip. Also, make sure you only use ONE patch *not* marked with a star at a time. As I wrote earlier, this is one of the technical limitations of the OPLL.

Pressing Escape will show a menu of pretty straight-forward operations. You can copy/swap patches using C and S and go back to the main editor using Q. You can also switch voice mode with M, which you should do now. You should now see a different patch editor, which is used for PSG sound. Pressing Escape > M again will bring you to the SCC patch editor, and pressing Escape > M once more will bring you back to the FM patch editor. It makes sense, right?

If you make changes to the patches, you have to save your new patch bank (or “voice file”). To do this, go back to the main editor, press Escape and enter the disk menu again and press 4 to “save voice”. Give it an appropriate name, and again, it has to be eight characters long. The next time you start the program, you will have to load both the song and the voice file you used/made for it to play back correctly. The file that gets loaded automatically when you start the program is START.VCD

I’ll leave it to you to figure out how to create and edit patches meaningfully through experimentation! You’ll get quite far using the standard patches, though.

Exporting and external players

Okay, so you’ve made a song and you want to play it without gaps in the looping and without having to load a voice file and music file every time. Enter the disk menu and press 5 to “Save BGM”. BGM is a format that is suitable for the computer to replay using a player program. It will ask you about what start address to use. I don’t know what works, but just press enter and use the suggested start address. Now the song will compile and calculate the end address, so don’t press anything. Then you will be prompted to name it. Again, the name has to be eight characters long.

The BGM file can be played in Winamp or XMPlay using the MSXplug plugin, or on the MSX using the Kinrou player. This takes some BASIC programming, but I've included an example player on "datafunk.dsk" if you want to give it a try.

Emulators and transfer tools

If you're going to use an emulator I suggest blueMSX. It emulates a lot of different models and peripherals accurately and you can insert a disk image simply by dragging the disk into the window.

As for transferring stuff to the MSX or copying files to/from a disk image, I suggest using Lex Lechz' Disk Manager (http://www.lexlechz.at/en/software/DiskMgr.html). You just need a PC with a disk drive to transfer stuff.

Notes of the differences between PAL and NTSC

There seems to be no pitch difference between playing a song on either system, but the tempo is much slower on PAL. The simple solution is to set the tempo by ear of course!

List of commands

A list of commands is available at http://www.nerlaska.com/msx/musica.html

Some of them are specific to the PC editor that is also presented on that page, like "{}n" and "&" but most of them should work in MuSICA.

Bye

I hope this has been at least somewhat helpful. If you have any questions or suggestions, feel free to mail me at **linde.philip@gmail.com**.