v0.2.1 alpha - Documentation

Tunes packed into a bunch of ASCII characters!

Written by: Tomáš "Frooxius" Mariančík

```
Warning: Project is still work in progress, any feedback,
suggestions and constructive cricism are welcome, as well as any
music created with it
```

Contents

Overview

This section provides general overview of the MusicString project, which should help you understand the general workings and capabilities that will be useful when studying the project thoroughly in the next chapter.

What is MusicString?

MusicString is a programming language for writing simple tones and music using just a few ASCII characters, allowing easy sharing of such tunes via text-based media like webpages, forums, micro blogging services (e.g. Twitter), IRC chats or Instant Messaging. The goal is to provide a simple way to write music and tunes with as few characters as possible, while keeping the syntax simple and straightforward.

What can it do?

Tunes and music are written as series of tones that are played in a sequence. Each tone is expressed with a single letter from the English alphabet, where each letter produces different frequency (pitch). MusicString is capable of playing theoretically unlimited number of channels at the same time, allowing creating chords or using several instruments at once.

Composer can choose from several instruments, where default set is composed from several basic soundwaves (sine, square, triangle, sawtooth, pluck and noise), however with use of soundfonts, it's possible to extend MusicString with any instrument/sound and combine them thanks to the ability to use virtually unlimited number of soundfonts at once.

Apart from pitch and instruments, it's also possible to change duration, volume and volume envelope (constant, fade in, fade out) of each tone. Language also provides constructs for reusing and repeating parts of the sequence and more advanced features like conditional execution.

Available software – how to use it?

There are several ways to create tunes with MusicString, although not all of them are currently implemented. Currently, you can use following software:

MusicString CLI compiler

Writing a MusicString program with this compiler is as simple as typing it in a plaintext file and then feeing it to the commandline compiler, which will produce a soundfile that can be played in regular audio players. The MusicString CLI compiler is always the most up-to-date version that supports the latest MusicString specification.

Alpha warning

MusicString is currently in alpha stage, which means that new features are rapidly implemented and existing ones can be removed or changed significantly, which means that songs you create now might stop working correctly and are going to need changes in order to work properly again. However, any testing, feedback and creations are highly appreciated and will help the project to get to more stable phase, were no such changes will be done, that would break compatibility.

Upcoming features

Following list contains yet-to-be implemented or documented features, planned for the MusicString projects.

- SoundFont support
- Vibrato
- Conditional execution
- Volume and instrument normalization
- MusicString visual player
- Mac OS X support
- Web browser support
- Compiler settings
- Setting note length by ratio

Syntax overview

This page serves as a quick reference when working with MusicString, for detailed description of listed constructs, please refer to the MusicString Syntax chapter.

- **a – Z** Each letter represents a tone of a different pitch
- **0 – 9** Changes duration of following tones
- **.** Pause – silent tone
- ~ Set instrument "~" (sine wave for default soundfont)
- **-** Set instrument "-" (square wave for default soundfont)
- **^** Set instrument "^" (triangle wave for default soundfont)
- **/** Set instruments "/" (sawtooth wave for default soundfont)
- **|** Set instrument "|" (pluck wave for default soundfont)
- ***** Set instrument "*" (noise wave for default soundfont)
- **(s0, s1, … , sn)**
  Fork – all statements from s0 to sn, begin execution at once, used for chords and channels
- **{name; body}**
  Defines a subroutine with given name and body
- **{name: a0, a1, … , an}**
  Subroutine usage with arguments a0 to an, arguments are optional
- **$n** Argument usage, **n** is the argument number in base 36
- **[body:repeats]**
  Anonymous loop that will repeat specified amount of times
- **[name:body:repeat]**
  Loop with specified name that will repeat specified amount of times
- **[name]**
  Stop all loops with given name
- **&n** Shift the pitch scale, n is alphabet letter determining shift
- **&+n** Shift the pitch scale, n is alphabet letter determining shift, plus one
- **&-n** Shift the pitch scale, n is alphabet letter determining shift, minus one
- **@bpm;** Set new tempo

- **#soundfont;** Set new soundfont
- **>** Set volume envelope to fadeout
- **<** Set volume envelope to fadein
- **=** Set volume envelope to linear
- **%n** Set the volume, n is alphabet letter determining the volume
- **?n** Skip over following statement if argument **n** is true
- **!n** Skip over following statement if argument **n** is false
- **?(n=x)** Skip over following statement if argument **n** equals **x**
- **?(n=x)** Skip over following statement if argument **n** doesn't equal **x**
- **?($n@y=x)** Skip over if result of operation equals x
- **?(y@$n=x)** Skip over if result of operation equals x
- **!($n@y=x)** Skip over if result of operation doesn't equal x
- **!(y@$n=x)** Skip over if result of operation doesn't equal x

MusicString syntax and usage

This section describes the syntax of the MusicString language and the way it works and generates sounds, allowing you to write your own tunes or implement your own tools.

Parsing Thread

MusicString uses something called a parsing thread. It's a virtual object that reads the MusicString sourcecode, character by character and produces a tone for every letter from English alphabet it encounters. It also holds currently selected instrument, tone duration, volume and other parameters. If it encounters a symbol that changes one of these parameters, it will change it immediately, applying the new setting to any of the following tones it produces.

Each Parsing Thread can fork into one or more child threads, each one having its own independent settings, which will perish when such child thread terminates, thread scope will however be discussed after the other syntax elements.

Thread properties

Each tone has a set of properties that determine the final sound produced, apart from pitch, which is determined by the letter used to generate the tone, all other properties are derived from the thread that generated given tone. Following properties are currently supported:

- Pitch/frequency
- Duration
- Volume/Amplitude
- Volume Envelope
- Instrument (one of 6 instruments)
- SoundFont
- Arguments

Whitespace

Any form of whitespaces, including regular spaces, tabs or newlines are ignored by the MusicString, so they can be used freely to better organize the code and increase readability. For compression purposes, they can be completely removed, since MusicString syntax doesn't rely on whitespaces.

Generating Tones

Every letter from English alphabet generates a single tone, where the position and case of the letter determines the pitch of the tone. Letters are organized by alphabet, from 'a' to 'Z', with alternating case of the letters – every lower case letter is followed by its uppercase version, which is in turn followed by lowercase version of the following letter in the alphabet. Therefore the series goes as follows:

```
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
```

Each letter has a parameter *n*, which is an integer number. Whole series is centered about the letter 'n', which has parameter *n* equal to zero. Every letter before it has parameter *n* lower by one, every letter after it has *n* higher than one. Therefore:

Letter	n	Frequency	Corresponding note
'a'	-26	98 Hz	G1
'n'	0	440 Hz	A4
'Z'	25	1864 Hz	A#6

Calculating the frequency

Using the parameter *n*, the frequency (pitch) of the tone is calculated using the following formula, which corresponds to the classical notes in music:

$$f = 2^{n/12} \times 440\ [Hz]$$

Setting tone duration

Tone duration is changed by using a single digit. All digits from 0 to 9 are mapped to specific note durations, digit 0 representing the shortest and digit 9 the longest one. Duration is changed for all tones following the digit during the scope of the given thread, unless it's changed again later. Length of notes corresponds to powers of two as in classical notes, for reference, use following table:

Digit	Note	Calculating length	Length for default bpm (120)
0	Hundred twenty-eighth	(1/(bpm/60)) / 32	15,625 milliseconds
1	Sixty-fourth	(1/(bpm/60)) / 16	31,25 milliseconds
2	Thirty-second	(1/(bpm/60)) / 8	62,5 milliseconds
3	Sixteenth	(1/(bpm/60)) / 4	125 milliseconds
4	Eighth	(1/(bpm/60)) / 2	250 milliseconds
5	Quarter **(default)**	(1/(bpm/60))	500 milliseconds
6	Half	(1/(bpm/60)) * 2	1 second
7	Whole	(1/(bpm/60)) * 4	2 seconds
8	Double whole	(1/(bpm/60)) * 8	4 seconds

9	Longa	(1/(bpm/60)) * 16	8 seconds

Changing tempo

As the table hints, actual length in seconds is based on beats per minute (bpm), which is 120 bpm by default. It is however possible to change the tempo using following syntax:

```
@bpm;
```

Here, *bpm* represents the new tempo – any floating point number. Tempo will change duration of all tones, beats per minute determines how many quarter notes there are in one minute of generated audio, therefore length of quarter note is determined by following equation:

$$t = \left(\frac{bpm}{60}\right)^{-1}$$

Length of other durations is determines either by multiplying or dividing this length by two, according to the table.

It is possible to change tempo at any time in the track, new tempo will be valid for all following tones in the scope of given thread, unless changed again. It is also possible to have several threads, each one with different tempo.

Shifting the frequency range

Because the basic range of the frequencies is somewhat limited, it's possible to perform a shift of the frequency range, which effectively extends the frequency range of MusicString and also provides a way to shift the frequencies of the tones, extending the supported frequency range.

Frequency shift is done by adding or subtracting a constant value from the parameter *n* before calculating the actual frequency in hertz. This can be expressed using parameter *m*. Its value is set to zero at the beginning and can be changed anytime during the execution of the program, making the shift valid for all following tones (unless changed again) processed by given thread.

Parameter *m* can be set by using symbol **&** followed by an alphabet letter:

```
&m
```

Letter **m** represents any symbol from the alphabet, whereas parameter *m* is calculated using the parameter *n* of each letter identical to generation of tones and multiplied by two.

$$m = ToneN(letter) * 2$$

Because this way will produce *m* that is always a multiply of two, additional versions of the frequency range shift statement exist that either add or subtract one from the *m* produced by aforementioned statement:

```
&+m
&-m
```

First version adds one to the *m*, second subtracts one. Therefore, following equations correspond to each version:

$$m = ToneN(letter) * 2 + 1$$
$$m = ToneN(letter) * 2 - 1$$

Taking the *m* parameter into the account, frequency of each tone is calculated according to following formula:

$$f = 2^{(n+m)/12} \times 440\ [Hz]$$

Thanks to this, the frequency range of MusicString is effectively extended to following range:

Letter	Shift	n	m	n+m	Frequency	Corresponding note
'a'	**&-a**	-26	-53	-79	4,59 Hz	D-3
'Z'	**&+Z**	25	51	76	35479,38 Hz	C#11

Inserting silence – pause

If composer needs to insert a period of silence into the composition, he can use period symbol for this:

.

Writing a single period will generate a silent tone, with the currently set duration of the thread that generated the tone. It can be thus considered a tone with zero volume.

It's important to say that this way generates silence only for the thread (channel) that reads the period symbol, if any of the other channels is playing any audible tones at the same moment, these tones will be still generated. In order to generate complete silence, all channels must generate silence at once. However, pause can be used for timing of channels that don't generate sound all the time.

Changing volume

If composer desires for some tones to be quieter or louder, he may change their volume. This is done by using following statement:

%v

Symbol *v* represents an alphabet letter from the same sequence used for generating tones. However, letter 'a' has parameter *n* equal to 0 and letter 'Z' has parameter *n* equal to 51. Volume is then calculated using following equation:

$$vol = \frac{letter}{51}$$

Therefore, volume ranges from 0.0 (silence) to 1.0 (maximum volume). Volume is valid for all following tones in the scope of given thread, unless changed again. It is possible to have several channels with different volumes.

Changing volume envelope

Apart from changing volume itself, it's possible to change also the volume envelope of each tone. By default, all tones have constant envelope – the volume doesn't change, however currently two

volume envelopes are supported: fade in and fade out, which can be used to modify sound of various instrument and generate interesting sound effects. Volume envelope can be changed by typing following symbols in the sequence:

Symbol	Envelope
=	Constant
>	Fade out
<	Fade in

Fade out and fade in envelopes apply to the whole tone duration, where volume starts/ends at the currently set volume and ends/starts at zero. The shape of the change of the envelope is quadratic.

Fade out and fade in envelopes work best with constant signals, instruments that already have fade in or fade out effect by default won't usually produce desirable effect. Fade out envelope can create "beats" from the instruments with constant (not fading) sound.

Setting instruments

MusicString supports several instruments that are used to play the tones and switching between them, which allows to change the characteristics of the produced sound. MusicString has six instrument slots, each one having its own symbol. By typing this symbol in the sequence, new instrument is set for all following tones (unless changed again).

Symbol	Instrument
~	Sine instrument
-	Square instrument
/	Sawtooth instrument
^	Triangle instrument
|	Pluck instrument
*	Noise instrument

Instruments are named after the shapes of the corresponding sound waves in the basic soundfont, for detailed information about them, please read the SoundFonts chapter.

Forking, Channels and Chords

Execution of the MusicString starts with a single parsing thread at the beginning of the string, which generates a single tone after tone. This is hardly satisfying for most music, therefore it's possible to fork this thread into several child threads, which generate tones in parallel. Using this, it's possible to create simple chords (playing several tones at once) or even several channels.

To fork thread, following syntax is used:

```
(s0, s1, ..., sn)
```

Parameters *s0* to *sn* represent sequences – any number of tones and other MusicString statements. They must be separated by commas. Once a thread encounters the fork statement, it creates a child thread for every sequence it finds (this doesn't include any possible sequences in nested fork threads) and then launches all of them at once and waits.

All child threads are guaranteed to launch execution at once, however they might not finish at the same time, depending on the length of each of the sequences. Once all child threads have finished execution, meaning that the last generated tone stops, the parent thread continues execution.

Inheritance and Thread properties scope

Each child thread inherits all Thread properties from the parent thread when it's created, however it can change any of them during the execution, without affecting any other threads (except the ones that it creates in case it encounters fork statement), including the parent thread.

Therefore, for example changing the instrument, or tone duration in a child thread doesn't change these properties for the parent thread, so once parent thread resumes execution, it will have the same Thread properties it had before it created the child threads.

Thanks to independent properties of each thread, It's possible to create several channels, each one with different tone duration or different instruments, that will all play at once.

Loops

In order to shorten the MusicString code, it's possible to repeat portions of the sequence using a loop statement. There are two types of loops: anonymous and named. Each loop needs to have specified number of repeats, using a positive nonzero integer. By using zero as the number of repeats, the loop will repeat infinitely. Loop is equivalent to writing sequence of the MusicString code several times in a row.

Anonymous loop

Simplest form of loop is anonymous one, which has following syntax:

```
[sequence : repeats]
```

This loop will repeat the *sequence* by a number of times specified after the semi-colon. It is possible to create an infinite loop by using zero as number of repeats, however there's no formal way of stopping it if needed.

Named loop

Any loop can be named, syntax is very similar to the anonymous loop, name of the loop is simply put at the beginning:

```
[name : sequence : repeats]
```

Name can contain any alphanumeric symbol and doesn't need to be unique. This loop works exactly the same way as the anonymous loop, except it can be stopped anytime from any other thread, including the infinite loops.

This allows creating a loop that will repeat unspecified amount of times (it's not required to manually calculate how many times it will repeat) and will be stopped when needed.

Stopping loop

It is possible to stop any currently running named loop by using following syntax:

```
[name]
```

Simply specifying a name of loop, without any sequence or number of repeats will send a termination signal to all loops with given name. However, loops won't terminate immediately, but only after they finish. If there's no loop with given name, nothing will happen.

Loops and threads

Once a thread encounters a loop, it creates a single child thread and waits, till it finished execution. This means, that this child thread inherits all Thread properties, but in case it changes any of them inside the loop body, it won't affect the parent thread, therefore once the loop finishes and thread continues, it will have same Thread properties before the loop began.

Subroutines

Loops allow to repeat portions of the tones only as a sequence, however to reuse portions at various places, subroutines have to be used. Subroutine is defined only once, but can be used as many times as needed. Additionally, it's possible to pass information into subroutines via arguments.

Subroutine definition

Subroutine is defined only once, usually at the beginning or at the end of the MusicString and it shouldn't be nested in any other subroutine definition or usage, it can be nested in a loop or fork, however there is no reason to do so and it will only clutter the code. Subroutine can be defined by following syntax:

```
{name; body}
```

Each subroutine must have unique name, which can contain any alphanumeric symbol and a body, which can contain any valid MusicString syntax, except subroutine definitions. It is recommended to place subroutine definitions either at the beginning or at the end of the MusicString code.

Subroutine usage

Any defined subroutine can be used at any place of the code using following syntax:

```
{name: a0, a1, ..., an}
```

It is possible to pass arguments to the subroutine, each subroutine usage can have up to 36 arguments, separated by comma. Each argument can consist of any MusicString code. Subroutine can be used even without arguments, however the semicolon still needs to be present.

Each time a thread encounters subroutine usage, it will create a child thread that will process the whole subroutine body in the subroutine definition. In case there are any arguments specified, the parent thread will write these arguments to the properties of the child thread, however all other properties, including the arguments that aren't set, are inherited. Once the child thread finishes execution of the subroutine body, the parent thread will continue execution after the subroutine usage statement.

Arguments

Each thread has 36 arguments, which can contain any valid MusicString code. The first thread starts with empty arguments. It is possible to set them by calling a subroutine with arguments. Whenever an argument is used, its body is placed at the point, where it's used. Arguments can be used as following:

```
$i
```

Parameter *i* is argument index, which can be any alphanumeric character, representing digits from base 36, starting with **0** and ending with **Z**. All letters representing digits must be uppercase.

Argument indirection

It is possible to pass argument index in another argument and then use this argument by following syntax:

```
$$i
```

Parameter *i* here, is index of the argument that contains a character representing index of the argument that is going to be used.

Changing Soundfont

By changing the SoundFont, it's possible to use a different set of instruments in the generated audio, the new instruments will replace the currently set instruments within the scope of given thread. SoundFont can be changed by following syntax:

```
#name;
```

Name represents either name of a built-in soundfont or a name of the file that represents given soundfont. New soundfont becomes valid for all following notes in the scope of given thread, unless changed again. Similarly to other statements, it's possible to change soundfont at any point in the code and to have different soundfonts in different threads. In combination with ability to have theoretically unlimited number of threads, number of used soundfonts at the same time is unlimited as well.

For details about integrated soundfonts and creating your own, please read the SoundFonts chapter.

SoundFonts

SoundFonts are the way of extending MusicString with virtually any sound and instrument. Thanks to the ability to use several soundfonts at once, it's possible to use any amount of instruments in a single MusicString code.

Integrated soundfonts

MusicString has several integraded soundfonts, these are implemented directly into the compiler and don't require any additional files. Each integrated soundfont has a name represented by a single letter, in addition to its formal name. Both shortened and formal names cannot be used by any external soundfonts, because compiler will automatically load the internal ones and ignore any external files with these names.

0 – "default"

This is the first and default soundfont, that's activated at the beginning, it contains basic soundwaves, that correspond to the names of the instruments:

- Sine instrument – sine wave, constant
- Square instrument – square wave, constant
- Triangle instrument – triangle wave, constant
- Sawtooth instrument – sawtooth wave, constant
- Pluck instrument – resembles sound of a drum, beat
- Noise instrument – generates noise – random signal
  This signal is generated by adding a randomly generated value to the noise value (that is confined to range from -1.0 to 1.0) every step, where the frequency of the tone determines how often is the randomly generated value changed.

External Soundfonts

Not implemented yet, sorry…

About

This section contains information about important contributions to the MusicString project. For up to date information about the project, please visit official website:

http://musicstring.solirax.org

MusicString Author

MusicString idea, design, documentation and official tools are created by:
Tomáš „Frooxius“ Mariančík

Contact information:

Personal website: http://frooxius.solirax.org

E-mail: tomas.mariancik@gmail.com
MSN/WLM/Yahoo: tomas.mariancik@hotmail.com
AIM: tails.cpp@solirax.org
GTalk: tomas.mariancik@gmail.com
Skype: frooxius
Twitter: frooxius

Thanks to people

I would like to thank to following people for help with the MusicString project:

- **1llusion** – for advices, valuable feedback and other suggestions based on his musical experience and studies (also for patience when I was blabbing about the project all the time)
- More coming… (need a permission first)