MPL

Music Processing Language

White Paper

Project Manager:	Bo Wang (bw2450)
Language Guru:	Yilin Xiong (yx2274)
System Architect:	Shengyi Lin (sl3759)
System Integrator:	Ying Tan (yt2443)
Verification and Validation:	Mengting Wu (mw2987)

Introduction

MPL is an excellent tool for our users: professional composers and music amateurs to compose their own music programmatically. Since contemporary music is often based on simple chord progressions, it can be coded easily by using well-designed data structure, operations and functions. Five basic data structures of int, note, melody, track, and music combined with their operations and functions are provided for recording timestamp, musical notation edition, melody manipulation, timbre modulation and music production. Besides, MPL also provides some advanced functions, such as automatically chord creation for notes and automatically harmony creation for melodies to make music more interesting and attractive. Also, certain pattern of melody can also be set as an independent sequence for re-use, making music composition more efficiently.

Motivation

Getting bored of recording your inspiration note by note on the paper? Want to know how does your music sound for some specified instrument directly? Instead of creating monotonous rhythm, want to create more attractive music? MPL is aimed at providing a non-traditional way for professional musicians or even beginners to create their own music easily and conveniently. Compositions can be completed with dynamics, arpeggios, accelerando, staccatos, repeats, and much more by simply using multiple libraries of existing data structures, operations and functions provided or create your own programmatically. Compose, record and arrange your song programmatically, your creativity can simply go wild by using a language like this!

Target Users

MPL is intended for both music amateurs and professional musicians, who have a passion for both music and programming. It allows you to create and process music in terms of notes, melody track or the entire piece of music in a convenient and interesting way.

If you are music amateurs or fans with merely basic music knowledge, we will provide you with libraries of existing data structures, operations and functions to compose your own music or conduct any level of recreation, including making adaptations of music with its pitch, timbre, chords, melody, tracks or the synthesis of different pieces of music. With MPL, you are free to immerse yourself into the world of music without the concerns of being an expert in both music and programming, and enjoy yourself to the ultimate.

For the music experts, on the other hand, you will be involved as both the users and the re-creators of MPL. With professional knowledge of music and a passion for programming, you can create your own functions with MPL to develop customized musical tools for composition. By such means, your creative ideas can also facilitate and reach users of MPL who share the common interest in music.

Sparking Points

MPL helps users to create notes, tracks, melodies as well as music. What's more, users can do arithmetic on data structures to synthesis and make adaptation of music. Compared to many music languages, MPL simplifies the data structure to make it easy to use. So, users can do music edition, composition more conveniently in a programming approach.

MPL is designed to be extendible, which allows composers and music software developers to build their own functions by programming. So, they can apply their professional musical knowledge in it, and create their own musical tools, compositions and instruments.

MPL also includes some advanced functions, which can intelligently identify the chords of a melody which users provide, and help users to improve the complexity of composition so they can do professional music arrangement for a simple melody without needing to know anything about music. Such as, recommendation of the harmonies chords to the simple melody, swapping the instruments, varies the choruses. As long as you know the basic melody you want to convey, our language can algorithmically create a customized, unique piece of music, synced to your melody, all in under a minute.

Features

MPL turns music composition and editing into operations like plus and minus. Programmer could also develop functions to create certain pattern of music. MPL goal is to make music calculable and programmable.

1. In MPL, music are calculated in 5 basic data structures: **Int**, **Note**, **Melody**, **Track** and **Music.** The following table summaries the definition and basic operations and of these data structures.

Int	Numeric variables holding whole numbers.
Note	Musical notation to represent a sound, including

	duration, pitch, starttime and strength.
Melody	Succession of notes.
Track	Container of melodies with a certain timbre.
Music	Several tracks construct a complete piece of music.

Table 1. Definitions of data structures

Note +/- Int	Raise/lower the pitch of Note by the value in Int.
Note + Note	Combine these Notes, and become to Melody
Melody +/- Note	Include/delete a Note into/from a Melody
Melody + Melody	Concatenate these Melodies into a new Melody
Music + Music	Concatenate these Music into a new Music
Melody * Int	Repeat the Melody for Int times to a new Melody
Melody * Melody	Superimpose Melodies to a new Melody
Music * Music	Superimpose all Tracks in these Music into a new Music
Melody	setDefault(): Set for default Notes in this Melody
Track	setTimbre(): Set Timbre for the Track
Music	read(): Read midi file and convert to Music type write(): Write Music type into a midi file

Table 2. Basic operations and functions for each data structures

2. User-defined function, if condition and for/while loop are supported in MPL -- an example is shown in Example6, which makes users develop more flexible programs to edit music.

3. MPL provides some advanced functions to edit music. For example, Harmony function adds harmony to a melody. Chord function generate required chord based on a note.

Representative Programs

Example1: Sample Syntax to create a piece of music.

```
void main(){
        // Initialization: create a piece of music with 3 tracks.
        // The first track has timbre of piano, both of the second
        // and the third tracks has timbre of violin.
        Int numOfTracks = 3;
        Track tracks[] = {PIANO, VIOLIN, VIOLIN};
        Music msc1(numOfTracks, tracks);

        // Create melody: C | D | E | F | G | A | B
        Int strength = 100;
        Int lasting = 200; // millisecond
        Int startTime = 0;
        Melody melody;

        melody += Note(C, startTime, strength, lasting)
                  + Note(D, startTime+lasting, strength, lasting);
        melody.setDefault(strength, lasting, ONE_BY_ONE);
        melody += Note(E) + Note(F) + Note(G) + Note(A) + Note(B);

        // Insert the melody to the first track at 100 millisecond
        // from the start of music
```

```
            Int timeStamp = 100;
            msc1.track[0].insert(timeStamp, melody);
        }
```

Example2: Sample Syntax of melody multiplication.

```
    void main(){
        Note notes1[] = {C, D, E};
        Note notes2[] = {E, F, G};
        // mld1: C | E | G, mld2: E | F | G
        Melody mld1(notes1);
              Melody mld2(notes2);
        // mld: C | D | E
        //      E | F | G
        Melody mld = mld1 * mld2;
    }
```

Example3: Sample Syntax of melody addition.

```
    void main(){
        Note notes1[] = {C, D, E};
        Note notes2[] = {F, G, A, B};
        // mld1: C | D | E, mld2: F | G | A | B
        Melody mld1(notes1);
        Melody mld2(notes2);
        // Concatenate two melodies: C | D | E | F | G | A | B
        Melody mld = mld1 + mld2;
    }
```

Example4: Sample Syntax of music multiplication.

```
        void main(){
            Music msc1 = read("music1.midi");
            Music msc2 = read("music2.midi");
            // music.midi will contain all tracks of music1.midi
            // and all tracks of music2.midi and all the melodies
            // in each track have the same start time 0
            Music msc = msc1 * msc2;
            write(msc, “music.midi”);
        }
```

Example5: Sample Syntax of music addition.

```
void main(){
    Music msc1 = read(“music1.midi”);
    Music msc2 = read(“music2.midi”);
    // music.midi will contain all tracks of music1.midi
    // and all tracks of music2.midi but the start time of
    // the melodies in the tracks of music2.midi will be
    // shifted to the end time of the melodies in the tracks
    // of music1.midi
    Music msc = msc1 + msc2;
    write(msc, “music.midi”);
}
```

Example6: Sample Syntax of writing a function to change the timbres of all tracks of music to the timbre of violin.

```
void setAllTimbresToViolin(Music &msc){
    // Use for loop to change the timbres of all tracks to
    // the timbre of violin
    for(Int i = 0; i < msc.numOfTracks(); i++)
            msc.track[i].setTimbre(VIOLIN);
}
```