**Analysis**

Think of product name

Put table in design of functions and their purposes in design

**Background to Problem**

The MIDI file format is a widespread and established standard for storing music. The music macro language (MML) is a niche language for describing music in a comparable way to how music is stored in MIDI files. It has no real standards, but has evolved over the years into different versions, the use of which are provided by BASIC implementations. One of the key issues with the MML is how there is not much software available that can play it; another issue is that the MML is, by design, only suited for monophonic tracks.

This project seeks to tackle both of these issues by providing a facility that can convert a number of MML files into a single multi-channel MIDI file. This allows the benefits of both the MML and MIDI file format to be taken advantage of – music is easily entered with MML, and it is easily playable using the MIDI format.

**Research Methods**

**The MIDI Specification**

The first thing I have researched for this project is the specification of MIDI files.

Values in MIDI files are either stored as big endian binary numbers or as big endian variable length quantities. Describe variable length quantities?

MIDI files are composed of “chunks”, which all consist of, at least, a 4-byte string, identifying the type of chunk, and the length of the chunk as a 32-bit integer. There are two different types of chunks featured in the MIDI specification 1.1 – header chunks and track chunks.

The following table shows how headers chunks are structured:

|  |  |  |  |
| --- | --- | --- | --- |
| Bits 0-7 | Bits 8-15 | Bits 16-23 | Bits 24-31 |
| The ASCII characters “MThd” | | | |
| Length | | | |
| Format | | Ntrks | |
| Division | |

“Length” is a 32-bit unsigned integer describing how many bytes are left of the chunk. It is always 6 for a header chunk.

“Format” is a 16-bit unsigned integer that describes the format of the MIDI file. It can currently take three values:

|  |  |
| --- | --- |
| 0 | The MIDI file is composed of a header chunk and a single track chunk |
| 1 | The MIDI file is composed of a header chunk  and multiple track chunks that are to be played simultaneously |
| 2 | The MIDI file is composed of a header chunk  and multiple track chunks that are to be played independently |

“Ntrks” is a 16-bit unsigned integer that indicates how many track chunks there are in the file.

“Division” is a 16-bit unsigned integer that sets the meaning of the delta-times. Describe delta-times?

The next table shows how track chunks are structured:

|  |  |  |  |
| --- | --- | --- | --- |
| Bits 0-7 | Bits 8-15 | Bits 16-23 | Bits 24-31 |
| The ASCII characters “MTrk” | | | |
| Length | | | |
| MTrk Events ... | | | |

“Length”, like in the header chunk, is a 32-bit unsigned integer describing how many remaining bytes there are to be read in the chunk.

The rest of the track chunk consists of MTrk events, which are structured as follows:

<MTrk event> = <delta time> <event>

“Delta time” is a variable length quantity that represents the length of time to leave between this event and the last one.

“Event” can be one of three different varieties: MIDI, sysex and meta.

MIDI events are MIDI channel messages, such as note down and note up.

Sysex (system exclusive) events are messages that are sent via the MIDI file, but are not related to the playing of it.

Meta events describe how the MIDI commands should be played, and give supplementary information about the track, such as its name and cues.

The MIDI events that will be relevant for the project are shown in the table below: “nnnn” is where the nibble representing the channel number the message is for is placed.

|  |  |  |
| --- | --- | --- |
| Status Byte | Data Bytes | Description |
| 1000 nnnn | 0kkkkkkk | Note off event – sent when a key is released.  0kkkkkkk is the note number.  0vvvvvvv is the velocity. |
| 0vvvvvvv |
| 1001 nnnn | 0kkkkkkk | Note on event – sent when a key is depressed.  0kkkkkkk is the note number.  0vvvvvvv is the velocity. |
| 0vvvvvvv |
| 1100 nnnn | 0ppppppp | Patch change – sent when the instrument is changed.  0ppppppp is the new patch number. |

The meta events that will be used in the project are shown below.

|  |  |
| --- | --- |
| Form (hex) | Description |
| FF 03 [length] [name] | Track name – this command contains the name of the track.  [length] is the length of the name following, as a variable length quantity.  [name] is an ASCII string. |
| FF 2F 00 | End of track – always present at the end of track chunk. |
| FF 51 03 tttttt | Set tempo.  “tttttt” is three bytes that represent the number of microseconds per MIDI crochet. |
| FF 58 04 nn dd cc bb | Time signature.  “nn” is a byte representing the top of a traditional time signature.  “dd” is a byte representing the bottom of a traditional time signature as .  “cc” is a byte setting the number of MIDI clocks per metronome tick.  “bb” is a byte setting the number of demisemiquavers per 24 MIDI clocks (usually 8) |

A full example of a MIDI file in hexadecimal is shown below, split up into a table with descriptions on each section.

|  |  |
| --- | --- |
| 4D 54 68 64 | “MThd” |
| 00 00 00 06 | Length of header chunk |
| 00 00 | Format 0 |
| 00 01 | One track |
| 00 20 | 32 ticks per crochet |
| 4D 54 72 6B | “MTrk” |
| 00 00 00 2A | Length of track chunk |
| 00 FF 58 04 04 02 18 08 | Set time signature to 4/4.  Set 24 MIDI clocks per metronome tick.  Set 8 demisemiquavers per 24 MIDI clocks |
| 00 FF 51 03 07 A1 20 | Set tempo to 0x07A120 (500 000) microseconds per 24 MIDI clocks. This corresponds to 120 BPM in this context. |
| 00 C0 30 | Set channel 0 to patch number 48 |
| 00 C1 30 | Set channel 1 to patch number 48 |
| 00 90 45 7F | Play note with MIDI number 69 (A4) on channel 0 with maximum velocity. |
| 81 10 80 45 7F | Release note with MIDI number 69 (A4) on channel 1 with maximum velocity after a delta time of 0x90. |
| 00 91 40 7F | Play note with MIDI number 64 (E4) on channel 1 with maximum velocity. |
| 7F 80 40 7F | Release note with MIDI number 64 (E4) on channel 1 with maximum velocity after a delta time of 0x7F. |
| 00 FF 2F 00 | End of track |

**Development Tools and Method**

There are many programming languages that would be suitable for this project, but I have chosen to go with C, as I am comfortable with it, and I feel that it is suitable, being a low-level language and the amount of byte-wise manipulation needed in this project.

Python would also have been a good choice because of how easy string manipulation is in it, which would make the processing of the input language more convenient.

I found two tools in my research that I have decided to use in this project to produce the parsing portions of the code – Lex and Yacc. Lex takes a C file containing regular expressions and code to execute upon matches, and Yacc, which works in conjunction with Lex, takes a C file containing Back-Naur form grammar and code to execute upon reductions.

To learn how to use Lex and Yacc, I’ve purchased an O’Riely book on them, which I’ll be using throughout the project.

An alternative to using Lex and Yacc would be writing my own parser and lexer, which could potentially yield faster code, but would take a great deal longer to develop.

To compile my code, I will use makefiles in combination with GCC. This is a common practice for C development, and one I am comfortable with. The alternative is entering the GCC, Lex and Yacc calls every time compilation is performed, which could be very inconvenient.

Clang is another C compiler that could be used, which has many advantages over GCC, such as compilation speed, but I am used to using GCC and have it already available on the machines I will be developing the project on.

**Description of the Current System**

The current way of converting multiple music macro language like files into a single multi-channel MIDI file is to step through by hand and enter the information as if writing a new MIDI file. This is tedious, unreliable and slow.

**Identification of Potential Users**

There is a small, but dedicated, remaining group of musicians that still use the music macro language

**Identification of User Need and Limitations**

Users will need to be able to use all of the syntax of the MML, as it would be difficult to enter music if even one feature was removed.

The users should not need to have any technical knowledge to use the solution.

The solution should be fast enough to keep up with a musicians work flow, otherwise this would be stifling for their creativity.

The syntax of the MML used by the solution should be at least similar to what already exists, so as to minimise the effort required by the user.

Fake user interaction, meant to be a product

**Data Flow Diagrams**

The following flow chart shows how the solution to the problem should work in terms of file flow:

**F:\School work\A Level\Computer Science\MML-To-Midi-Project\Project Documentation\Diagrams\Analysis_Data_Flow.png**

**Use Cases**

A musician whom uses the MML to compose music would likely find the tools written for this project very useful, as it allows the conversion of their preferred format to an easily playable and portable form.

The solution would also very suitable for a less musically experienced individual, as it is easy to alter and enter music in MML, in contrast to traditional sheet music.

**Potential Solutions**

One solution to the problem is to have one terminal program that the user enters the paths to the input MML files into, and have it output the desired MIDI file. There are a number of pros and cons in this potential solution:

|  |  |
| --- | --- |
| Single Program Advantages | Single Program Disadvantages |
| Easy to use | The program would be complex, given what functionality it must have |

Another solution would be to have two programs that are designed to be used in conjunction: the first would take a single MML file and output a single-channel MIDI file, and the second would take multiple single-channel MIDI files and output a single multiple-channel MIDI file. Some advantages and disadvantages to this approach are shown below:

|  |  |
| --- | --- |
| Program Pair Advantages | Program Pair Disadvantages |
| Each program would be more simple to write, compared to a single complex program | More difficult for to use |
| Each channel of the final MIDI file can be listened to, as a single-channel MIDI file of it would be generated before-hand |  |

**Chosen Solution**

I have chosen to use the solution featuring a pair of programs. This is primarily because I feel that the complexity of both programs combined would be less than the complexity of one larger program.

**Other Evidence**

Research that doesn’t fit into any other categories.

**Objectives of the Project**

* A program should be written that takes text file containing a variant of the MML as an input, and outputs a single-channel MIDI file that can be play with conventional software.
* A program should be written that takes multiple MIDI files generated by the aforementioned program and combines them into a single multi-channel MIDI file. This combined MIDI file should be playable using conventional software also.
* The programs written for this project should:
  + Complete their execution in under one second, so as to not interrupt the users work flow
  + Use less than half a megabyte of memory during execution
  + Be a less than a quarter of a megabyte in size
* A version of the music macro language should be designed that will be used as the input for the program that generates a single-channel MIDI file from a single MML file. This language should:
  + Have all the functionality of existing variants of the MML, including support for:
    - Octave changing
    - Accidentals
    - Default length setting
    - Volume changing
    - Tempo setting
    - Macros
  + Have as unambiguous a syntax as possible, with a clear logical progression

Bibliography