**Documented Design**

Fully or nearly fully articulated design for a real problem, that describes how all or almost all of the key aspects of the solution/investigation are to be structured/are structured.

**The Problem and Solution**

The problem is to combine multiple music macro language text files into a single multi-channel MIDI file. My chosen approach to this problem is to write two terminal programs that would be used in conjunction. The first program, named “mmltomidi”, would take a single MML file and output a single single-channel MIDI file. The second program, named “catmidi” would take multiple single-channel MIDI files and combine them to create a single multi-channel MIDI file.

**Target Hardware**

The programs are will be written for Unix based operating systems such as OSX and Linux, as this is what I will be developing them on, and I am also used to working with these systems. Also, the programs will be written to run on little endian processors – this is relevant because the endianness of many values are flipped in the programs.

**Development Environment**

Both the programs will be written in C, and mmltomidi will use the lex and yacc compiler compilers (this is described more in the technical solution.) They will be compiled using the GCC, which will be called by hand written makefiles. These makefiles will also handle the enabling of debugging, and the clearing of build directories.

**Overall System Design**

The following tables are IPSO charts for the mmltomidi and catmidi programs respectively:

|  |  |  |  |
| --- | --- | --- | --- |
| Inputs | Processes | Storage | Outputs |
| MML text file | MML text | Single-channel MIDI file | Success message |

|  |  |  |  |
| --- | --- | --- | --- |
| Inputs | Processes | Storage | Outputs |
| Single-channel MIDI files | MIDI file contents | Multi-channel MIDI file | Success message |

**mmltomidi – User Inteface**

The mmltomidi program will be called via the terminal with the form shown below:

mmltomidi [-o output\_path] mml\_file

The “-o” switch sets the output file to be the “output\_path” following. If the switch is not present, then the output file will be called “output.midi” and placed in the working directory. The “mml\_file” portion is where the path to the input file is put.

**catmidi – User Inteface**

The catmidi program will be called, similarly to mmltomidi, via the terminal with the form shown below:

catmidi [-o output\_path] [path ...]

The “-o” switch will function exactly as described in the mmltomidi user interface section. “[path ...]” is where the paths to the input MIDI files are put, delimited by spaces.

ß

**mmltomidi – Procedural Abstraction**

The following is a flow chart showing a broad abstraction of how the mmltomidi program will work:

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

**mmltomidi – Key Algorithms**

One of the most utilised algorithms will be to write a variable length quantity. It will take a pointer to where the data should be written, and the number that should be written as a variable length quantity. It will return the length of the data written. The algorithm I will use for this is shown below as a C function.

int writeVariableLengthQuantity(char \*dest, unsigned int n) {

if (n == 0) {

\*dest = 0;

return 1;

}

int length = sizeof(int);

for (int i = sizeof(int) - 1; i >= 0; i--) {

if (n >> i \* 7) {

break;

} else {

length--;

}

}

for (int i = length - 1; i >= 0; i--) {

if (i == 0) {

\*(dest + length - i - 1) = (n >> i \* 7) & 0x7F;

} else {

\*(dest + length - i - 1) = ((n >> i \* 7) & 0x7F) + 0x80;

}

}

return length;

}

An endianness swapper is another algorithm that will be commonly used throughout the mmltomidi program. It will take a value as an input, and output the same value, but with it’s endian swapped. A version for swapping an integers endianness it is shown below as a C function.

int swapIntEndian(int n) {

int o = 0;

for (int i = 0; i < sizeof(int); i++) {

\*((char \*) &o + sizeof(int) - i - 1) = \*((char \*) &n + i);

}

return o;

}

The key algorithm of the program, however, is that which generates the MIDI data from the processed MML text data. The flowchart below demonstrates how it will function.

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

**mmltomidi – Main Data Structures**

The following chunk of code represents the key data structures that store the MML data found by the parser for the MIDI data generating program.

struct note {

char command; //Letter

char accidental; //-1 for flat, 1 for sharp

unsigned char modifier; //Number after

};

struct mmlFileStruct {

char name[256]; //Null terminated

struct note notes[16384]; //Observe size limit

int noteCount;

};

The main structure is “mmlFileStruct”. This contains the name of the track, the number of notes in the track, and a list of every note and command in the program. The “note” structure is used only to represent a note or command in the “mmlFileStruct” structure. It contains the letter representing the command, whether a note is an accidental or not, and the modifying number for the command.

**catmidi – Procedural Abstraction**

The following is a flow chart showing a broad abstraction of how the catmidi program will work:

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

**catmidi – Key Algorithms**

The catmidi program shares many of the same algorithms as the mmltomidi program, including the variable length quantity writing algorithm and the endianness swapper. In light of this, only algorithms which aren’t featured in the mmltomidi program are described below.

One of the main algorithms used in the catmidi program is one that reads a variable length quantity and returns an integer. This is very similar to the algorithm that writes a variable length quantity featured in a previous section. It will take a pointer to the variable length quantity to be read as an input, and return an integer. The algorithm is shown below as a C function.

int readVariableLengthQuantity(char \*ptr) {

char \*originalPtr = ptr;

while (\*(ptr) & 0x80) {

ptr++;

}

int output = 0;

int outputShift = 0;

do {

output |= (\*ptr & 0x7F) << outputShift;

outputShift += 7;

} while (ptr-- != originalPtr);

return output;

}

A core algorithm used is that which reads an event from an input MIDI file and outputs a structure containing the event and its delta time, with the channel number replaced where appropriate. It will take a pointer to the buffer where the MIDI data is stored, a pointer to a event structure, which is where the event will be stored, and a channel number that will replace the existing channel number in some commands. The algorithm is shown on the following page as a C function.

void readMTrkEvent(unsigned char \*input, struct mtrkEvent \*outputPtr, char channelNumber) {

outputPtr->deltaTime = readVariableLengthQuantity((char \*) input);

while (\*input & 0x80) {

input++;

}

input++;

unsigned char \*originalInputPtr = input;

switch (\*input) {

case 0xFF:

input++;

switch (\*input) {

case 0x03: //Name

input++;

input += \*input + 1;

break;

case 0x2f: //End

input += 2;

break;

case 0x51: //Tempo

input += 5;

break;

case 0x58: //Time sig.

input += 6;

break;

default:

//Unknown command, error here

break;

}

break;

case 0x80: //Note off

case 0x90: //Note on

\*input |= channelNumber;

input += 3;

break;

case 0xC0: //Patch change

\*input |= channelNumber;

input += 2;

break;

default:

//Unknown command, error here

break;

}

outputPtr->length = input - originalInputPtr;

memcpy(outputPtr->event, originalInputPtr, outputPtr->length);

}

**catmidi – Main Data Structures**

The only data structure in use in the catmidi program is that which stores the read MIDI events. It is shown below as a C structure.

struct mtrkEvent {

char event[262];

short length;

int deltaTime;

};

The event itself is stored in “event”, which is sized such that no recognised command will exceed its capacity. The length of the event is stored in “length”. Finally, the delta time of the MIDI event (that is, the time between carrying out the following command and the previous one), is stored in the “deltaTime” integer. In the MIDI file itself this is stored as a variable length quantity.

**Music Macro Language Design**

This section describes the music macro language used by the mmltomidi program.

**Introduction**

The music macro language (MML) is a music description language that has been in use since 1978, although this was an early version. There has never been an official specification, so each implementation varies slightly, and over the years the language has evolved. The MML to MIDI converter uses a version of the MML derived largely from “Classical MML” with some “Modern MML” features present. Some new specific commands are included also, and some commands are changed where necessary.

All commands in this language have their own line and are terminated by a new line (“\n”, “\r “or “\r\n”).

**Comments**

Comments are started with two hash characters at the beginning of a new line . This makes the remainder of the line a comment; any more hashes found on the line have no effect. Two hashes are used because single hash starts a meta command.

**Playing Notes**

The “play” command is used to play a series of notes and macros. Spaces can be intermingled with the notes to improve the clarity of the code. An example usage of this command is shown below:

play c5e5g5

**Note Syntax**

Notes are written as the note name followed optionally by the length of the note as a digit – each value for this digit represents a musical note length, which can be seen in the table below. If a length is not given, the default value is used, which is initially 5, but can be changed with the “l” command detailed shortly. A rest is represented by the note name “r”. To play an accidental note a “+” or “-”, respectively, is added after the note name and before the note length. Accidentals applied to rests are ignored.

|  |  |  |
| --- | --- | --- |
| MML Note Value Number | Musical Note | |
| American Notation | Name |
| 0 | 1/32 | Demisemiquaver |
| 1 | 1/16 | Semiquaver |
| 2 | 1/16 + 1/32 | Dotted semiquaver |
| 3 | 1/8 | Quaver |
| 4 | 1/8 + 1/16 | Dotted quaver |
| 5 | 1/4 | Crochet |
| 6 | 1/4 + 1/8 | Dotted crochet |
| 7 | 1/2 | Minim |
| 8 | 1/2 + 1/4 | Dotted minium |
| 9 | 1 | Semibreve |

To alter how each note is played, there are some of commands entered with the notes. These are listed below (where square brackets and their contents are not literal):

* o[digit] Set the octave each following note is played in. The digit represents the scientific pitch notation (SPN) number of the desired octave. All notes entered before this command is entered are played in the 4th SPN octave (“A” will be 440 Hz.)
* < Shift the octave down by one.
* > Shift the octave up by one.
* v[digit] Set the volume of the following notes. By default, notes will play at 100% volume.
* p[number from 0 to 11] Transpose all the following notes up by the number following ‘p’ semitones. The default setting is 0.
* l[digit] Set the default length of the following notes to the digit. The initial default length is 5. Note that this does not affect the ‘v’ or ‘o’ commands.

In modern MML there is also a “t” command, which sets the tempo. This is not included, as a more obvious command on it’s own line is favoured for ease of reading.

**Meta Commands**

These commands are entered on their own lines only once and are all preceded by a single hash. They tell the converter how the rest of the file should be played and add information to the MIDI file.

* #tempo [BPM] – set the tempo in BPM of the track (where a beat is a crochet.) This should be set the same in each MML track file when combining them into one MIDI file. The default tempo is 120 BPM.
* #instrument [general MIDI patch number] – set the instrument the rest of the file should be played with. The default instrument is a piano (GM patch number 0.) This command is not present in other MML versions because it is only useful if the file is being converted to a MIDI file.
* #name [name] – set the name of the track. This is put verbatim into the MIDI file in a track name meta event, and can be very useful when altering the MIDI file directly. Only one instance of this command should be in a MML file, otherwise a syntax error will occur.

**Macros**

A macro in this version of MML is written as below (on it’s own line):

$c v9o4c5

The dollar sign shows that this is a macro definition, and the letter following this is the “name” of the macro. The text after the dollar sign and letter replaces any other instance of the macro name found. A limitation of this notation is that there are only 26 possible macro names, but it is done this way to be more compatible with other versions of MML. Macros can be defined more than once.

**Full Example**

To conclude the section, a short example MML file is shown below.

|  |
| --- |
| ##Example comment  #name test\_track  #instrument 0  #tempo 120  $c l3o4cdefgab>c9  play v8$c |