**Documented Design**

**Target Hardware**

The programs are will be written for Unix based operating systems such as OS X and Linux, as this is what I will be developing them on and am 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 numbers will be flipped in the programs.

**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 Interface**

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 Interface**

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

**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

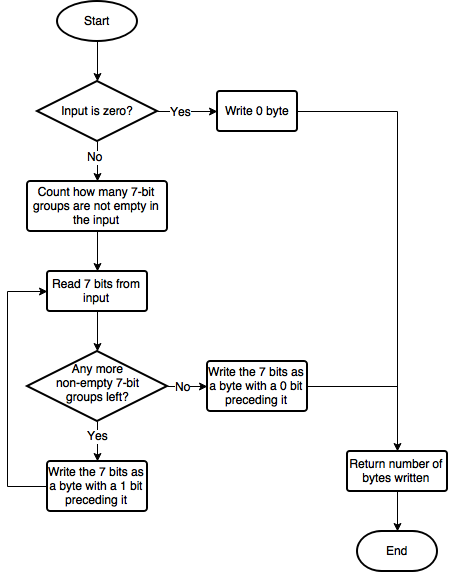
**mmltomidi – Key Algorithms**

**Variable Length Quantity Writing**

One of the most utilised algorithms will be to write a variable length quantity. It will take as inputs 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. A list of inputs and expected outputs in hexadecimal for this algorithm are shown below.

|  |  |  |
| --- | --- | --- |
| Input Number | Data Written | Length of Data Written |
| 00 00 00 00 | 00 | 1 |
| 00 00 00 7F | 7F | 1 |
| 00 00 00 80 | 81 00 | 2 |
| 00 00 3F FF | FF 7F | 2 |
| 00 00 40 00 | 81 80 00 | 3 |
| 0F FF FF FF | FF FF FF 7F | 4 |

A flow chart showing a procedural abstraction of the algorithm is below.



**Endianness Swapper**

An endianness swapper is another algorithm that will be commonly used throughout the mmltomidi program. It will take a binary number as an input, and output the same binary number, but with it’s endian swapped. The table below shows what outputs would be given for example inputs.

|  |  |
| --- | --- |
| Input Binary Number | Output Binary Number |
| 00 00 00 00 | 00 00 00 00 |
| 00 00 00 FF | FF 00 00 00 |
| 00 FF 00 00 | 00 00 FF 00 |
| 12 34 56 78 | 78 56 34 12 |

A C-like pseudo-code implementation of this algorithm is shown below.

|  |
| --- |
| int swapIntEndian(int input) {  int output = 0;  for each byte in input {  write the byte to the byte of opposite significance in output  }  return output  } |

**MIDI Data from Processed MML Commands**

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];

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.

Below is an example assignment of these data structures.

|  |
| --- |
| struct mmlFileStruct exampleMMLFileStruct;  strcpy(exampleMMLFileStruct.name, “Test track”);  exampleMMLFileStruct.notes[0].command = ‘a’;  exampleMMLFileStruct.notes[0].accidental = 0;  exampleMMLFileStruct.notes[0].modifier = 5;  exampleMMLFileStruct.noteCount = 1 |

This example data represents a MML file with the name “Test track”, which contains a single crochet of note “A”.

**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 in the following sections.

**Variable Length Quantity Reading**

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 variable length quantity writing algorithm covered before. It will take a pointer to the variable length quantity to be read as an input, and return an integer. The table below shows what outputs would be expected from the given inputs.

|  |  |
| --- | --- |
| Input Data | Number Outputted |
| 00 00 00 00 | 00 |
| 00 00 00 7F | 7F |
| 00 00 81 00 | 80 |
| 00 00 FF 7F | 3F FF |
| 00 81 80 00 | 40 00 |
| FF FF FF 7F | 0F FF FF FF |

An implementation of this algorithm in C-like pseudo-code is shown below.

|  |
| --- |
| int variableLengthQuantity(char \*inputPtr) {  int output = 0;  while (1) {  read byte from input  append first 7 bits from read byte to output  if read byte has bit 8 set {  continue;  }  break;  }  return output;  } |

**Read MIDI Event**

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. The algorithm is broadly demonstrated below in pseudo-code.

|  |
| --- |
| void readMTrkEvent(unsigned char \*inputPtr, struct mtrkEvent \*outputPtr, char channelNumber) {  outputPtr->deltaTime = readVariableLengthQuantity(inputPtr);    Copy event to outputPtr->event    Replace channel number in applicable events    outputPtr->length = length of event stored  } |

**Combine MIDI Files**

The main algorithm of the catmidi program is that which actually combines the MIDI files. It will take a pointer to an output buffer and pointers to MIDI files that have been read into memory. The pseudo-code for this algorithm is shown below.

|  |
| --- |
| int combineMIDIFiles(char \*outputBuffer, unsigned char \*inputBuffer[]) {  Write output chunk headers    Move the input buffer pointers to the start of events    Read an event from each MIDI file and store them  while (1) {  Write the event with the smallest delta time to the output    If this was the end of track MIDI event {  return length of written MIDI file;  }    Subtract delta time of event written from all other event delta times    Read the next event from the input buffer the event was written from  }  } |

**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.

An example assignment of this structure is shown below.

|  |
| --- |
| struct mtrkEvent exampleEvent;  strcpy(exampleEvent.event, (char []) {0xFF, 0x2F, 0x00});  exampleEvent.length = 3;  exampleEvent.deltaTime = 0; |

**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 minim |
| 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 its 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  $c l3o4cdefgab>c  play v8$c |

This example plays the C major scale using quavers at a BPM of 120. The equivalent of this file in sheet music is shown below.



**MML in Regex and BNF**

This section shows how I will encode the language described in the previous section using regular expressions and Back-Naur form. These expressions can be put verbatim into the Lex and Yacc input files. The following table shows what regexes correspond to what BNF symbol, and gives a description on it.

|  |  |  |
| --- | --- | --- |
| Regex | BNF Symbol | Description |
| ^##.\*(\r|\n|(\r\n)) | COMMENT | Matches comment lines. |
| ^(\r|\n|(\r\n)) | LINE\_BREAK | Matches an empty line. |
| ^#tempo" "[0-9]{1,3}(\r|\n|(\r\n)) | TEMPO\_SET | Matches a tempo setting line. |
| ^#instrument" "[0-9]+(\r|\n|(\r\n)) | INSTRUMENT\_SET | Matches an instrument assigning line. |
| ^#name" "[a-zA-Z0-9\_]+(\r|\n|(\r\n)) | NAME\_SET | Matches a name setting line. |
| ^$[a-z]" "(([cdefgabrov][+-]?[0-9]?)|  ($[a-z])|[<>]|(p[0-9]+)|(l[0-9])|" ")+  (\r|\n|(\r\n)) | MACRO\_ASSIGNED | Matches a macro assignment. The middle portion is the regex that matches series of notes. |
| ^play" "(([cdefgabrov][+-]?[0-9]?)|  ($[a-z])|[<>]|(p[0-9]{1,2})|(l[0-9])|" ")+(\r|\n|(\r\n)) | PLAY\_COMMAND | Matches a play command. The middle portion is, again, the regex that matches a series of notes. |

The figure below contains the BNF grammer for the MML. “mmlFile” is the start symbol.

|  |
| --- |
| <mmlFile> ::= <line> | <mmlFile> <line>  line ::= LINE\_BREAK |  COMMENT |  TEMPO\_SET |  INSTRUMENT\_SET |  NAME\_SET |  MACRO\_ASSIGNED |  PLAY\_COMMAND |