A412

0.1

Generated by Doxygen 1.8.8

Mon Dec 15 2014 08:29:04

Contents

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

2 File Index

2.1 File List

Here is a list of all files with brief descriptions:

3 Data Structure Documentation

3.1 data Struct Reference

Data Fields

- unsigned int **tempo**
- int **ppqn**
- **mode mode**

3.1.1 Detailed Description

A struct containing general data pertaining to the song

Parameters

tempo	the tempo in beats-per-minute
ppqn	ticks-per-quarter-note contains the number of ticks per quarter note
mode	an enumerated value representing the mode (major/minor)

3.1.2 Field Documentation

3.1.2.1 mode data::mode

3.1.2.2 int data::ppqn

3.1.2.3 unsigned int data::tempo

The documentation for this struct was generated from the following file:

- **main.c**

3.2 eventPlacement Struct Reference

Data Fields

- int **noteOn**
- int **noteOff**
- int **afterTouch**
- int **controlChange**
- int **programChange**
- int **channelPressure**
- int **pitchWheel**

3.2.1 Detailed Description

A struct containing placements of midi events, stored as their placement in file

Parameters

noteOn	signals when a note starts playing
noteOff	signals when a note stops playing
afterTouch	changes velocity for a single note on a single channel
controlChange	used for a large number of effects, none of which are used in this project (stored to find deltatimes)
programChange	signals instrument change (not used; stored to find deltatimes)
channelPressure	changes velocity for all notes on a specific channel (akin to a global afterTouch)
pitchWheel	fine tuning of pitch for all notes on a specific channel (similar to channelPressure, but for pitch)

3.2.2 Field Documentation

3.2.2.1 int eventPlacement::afterTouch

3.2.2.2 int eventPlacement::channelPressure

3.2.2.3 int eventPlacement::controlChange

3.2.2.4 int eventPlacement::noteOff

3.2.2.5 int eventPlacement::noteOn

3.2.2.6 int eventPlacement::pitchWheel

3.2.2.7 int eventPlacement::programChange

The documentation for this struct was generated from the following file:

- **main.c**

3.3 moodWeighting Struct Reference

Data Fields

- char **name** [25]
- int **mode**
- int **tempo**
- int **toneLength**
- int **pitch**

3.3.1 Detailed Description

A struct containing a single moods name and weighting

Parameters

name	the name of the mood
mode	a value -5 to 5 representing this parameters impact on the mood
tempo	a value -5 to 5 representing this parameters impact on the mood
toneLength	a value -5 to 5 representing this parameters impact on the mood

pitch	a value -5 to 5 representing this parameters impact on the mood

3.3.2 Field Documentation

3.3.2.1 int moodWeighting::mode

3.3.2.2 char moodWeighting::name[25]

3.3.2.3 int moodWeighting::pitch

3.3.2.4 int moodWeighting::tempo

3.3.2.5 int moodWeighting::toneLength

The documentation for this struct was generated from the following file:

- **main.c**

3.4 note Struct Reference

Data Fields

- int **tone**
- int **octave**
- int **length**
- int **average**
- int **ticks**

3.4.1 Detailed Description

A struct cotaining data about a single note

Parameters

tone	the tone stored as an integer (C = 0)
octave	which octave, on a piano, the note is in (1 is the deepest, C4 is middle C)
length	the notes length in standard musical notation
average	used in calculating the average tone
ticks	the notes length in ticks

3.4.2 Field Documentation

3.4.2.1 int note::average

3.4.2.2 int note::length

3.4.2.3 int note::octave

3.4.2.4 int note::ticks

3.4.2.5 int note::tone

The documentation for this struct was generated from the following file:

- **main.c**

4 File Documentation

4.1 main.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <dirent.h>
```

Data Structures

- struct **note**
- struct **data**
- struct **moodWeighting**
- struct **eventPlacement**

Macros

- #define **CHARS** 1000
- #define **SCALESIZE** 7

Typedefs

- typedef enum **mode mode**
- typedef enum **tone tone**
- typedef enum **mood mood**

Enumerations

- enum **mode** { **major**, **minor** }
- enum **tone** {
  C, **Csharp**, **D**, **Dsharp**,
  E, **F**, **Fsharp**, **G**,
  Gsharp, **A**, **Asharp**, **B** }
- enum **mood** { **glad**, **sad** }

Functions

- void **checkDirectory** (char ∗, DIR ∗)
- void **printNote** (**note**)
- int **getHex** (FILE ∗, int[])
- void **fillSongData** (**data** ∗, int[], int)
- int **countPotentialNotes** (int[], int)
- void **fillNote** (int, **note** ∗)
- void **settingPoints** (int ∗, int ∗, int ∗, int ∗, **data**, int, **note**[], int ∗)
- void **insertMoods** (**moodWeighting**[], FILE ∗)
- void **weightingMatrix** (**moodWeighting**[], int, int, int, int, int ∗)
- void **findEvents** (int, int[], **eventPlacement**[], **note**[], int ∗, int ∗)
- void **insertPlacement1** (int[], int ∗, int, **note**[], int ∗, int[])
- void **insertPlacement2** (int[], int ∗, int)
- int **checkNextEvent** (int[], int)

- void **findTicks** (int, int[], **eventPlacement**[], **note**[], int, int *, int[])
- void **countTicks1** (int[], int *, int, **note**[], int *)
- void **countTicks2** (int[], int *, int, **note**[], int *)
- void **deltaTimeToNoteLength** (int, int, **note** *)
- int **isInScale** (int, int[], int)
- int **isInMinor** (int)
- int **isInMajor** (int)
- int **sortToner** (const void *, const void *)
- void **findMode** (**note** *, int, **data** *)
- int **FindMoodAmount** (FILE *)
- void **printResults** (int, int, int, int, **moodWeighting**[], int[])
- int **main** (int argc, const char *argv[])
- int **sortTones** (const void *a, const void *b)
- void **checkScale** (int scales[], int **tone**, int key)
- void **findMode** (**note** noteAr[], int totalNotes, **data** ***data**)

Variables

- int **AMOUNT_OF_MOODS**

4.1.1 Macro Definition Documentation

4.1.1.1 #define CHARS 1000

4.1.1.2 #define SCALESIZE 7

4.1.2 Typedef Documentation

4.1.2.1 typedef enum mode mode

4.1.2.2 typedef enum mood mood

4.1.2.3 typedef enum tone tone

4.1.3 Enumeration Type Documentation

4.1.3.1 enum mode

Enumerator

major

minor

```
00026 {major, minor} mode;
```

4.1.3.2 enum mood

Enumerator

glad

sad

```
00028 {glad, sad} mood;
```

4.1.3.3 enum tone

Enumerator

C

Csharp

D

Dsharp

E

F

Fsharp

G

Gsharp

A

Asharp

B

```
00027 {C, Csharp, D, Dsharp, E, F, Fsharp, G, Gsharp, A, Asharp, B} tone;
```

4.1.4 Function Documentation

4.1.4.1 void checkDirectory (char * *MIDIfile*, DIR * *dir*)

A function to read music directory and prompt user to choose file

Parameters

MIDIfile	a pointer to a string containing the name of the chosen input file
dir	a pointer to a directory

```
00193                                                {
00194   struct dirent *musicDir;
00195   int musicNumber = -2;
00196
00197   if((dir = opendir ("./Music")) != NULL) {
00198     printf(" Mulige numre\n");
00199     while ((musicDir = readdir (dir)) != NULL){
00200       if(musicNumber > -1 && musicNumber < 10)
00201         printf (" %d.  %s\n", musicNumber++, musicDir->d_name);
00202       else if(musicNumber > -1)
00203         printf (" %d. %s\n", musicNumber++, musicDir->d_name);
00204       else
00205         musicNumber++;
00206     }
00207   }
00208   else{
00209     perror ("Failure while opening directory");
00210     exit (EXIT_FAILURE);
00211   }
00212
00213   closedir(dir);
00214
00215   if((dir = opendir ("./Music")) != NULL) {
00216     printf("\n Indtast det valgte nummer\n ");
00217     scanf(" %d", &musicNumber);
00218
00219     for(int i = -2; i <= musicNumber; i++)
00220       if((musicDir = readdir (dir)) != NULL && i == (musicNumber))
00221         strcpy(MIDIfile, musicDir->d_name);
00222
00223     printf("\n Du valgte \n %s\n Hvilket giver disse resultater\n", MIDIfile);
00224   }
00225   else{
00226     perror ("Failure while opening directory");
00227     exit (EXIT_FAILURE);
00228   }
00229
00230   chdir("./Music");
00231 }
```

4.1.4.2 int checkNextEvent (int *hex[],* int *j*)

```
00331                                       {
00332   switch (hex[j]){
00333     case 0x90:
00334     case 0x80:
00335     case 0xA0:
00336     case 0xB0:
00337     case 0xC0:
00338     case 0xD0:
00339     case 0xE0: return 1; break;
00340     default  : return 0; break;
00341   }
00342 }
```

4.1.4.3 void checkScale (int *scales[],* int *tone,* int *key*)

Checks if the tone given is within the scale of the key given.

Parameters

scales	An array containing the scalas
tone	An integer representing the tone to be checked
key	Integer representing the key the note is compared to

```
00611                                                 {
00612   if(tone < key)
00613     tone += 12;
00614
00615   scales[key] = isInMajor(tone - key);
00616 }
```

4.1.4.4 int countPotentialNotes (int *hex[],* int *amount*)

A function to count the number of notes in the entire song

Parameters

hex[]	an array with the stored information from the file
amount	an integer holding the total number of characters in the array

```
00252                                               {
00253   int i = 0, res = 0;
00254
00255   for(i = 0; i < amount; i++){
00256     if(hex[i] == 0x90){
00257       res++;
00258     }
00259   }
00260
00261   return res;
00262 }
```

4.1.4.5 void countTicks1 (int *hex[],* int * *i,* int *deltaCounter,* note *noteAr[],* int * *tickCounter*)

Processes events with two parameters, extracting deltatime (and advancing the file pointer)

```
00387                                                                                 {
00388   noteAr[*tickCounter].ticks = 0;
00389   int tick = 0;
00390
00391   while(deltaCounter < 7 && hex[(*i + deltaCounter)] > 0x80)
00392     tick += ((hex[(*i + deltaCounter++)] - 0x80) << 7);
00393
00394   tick += hex[(*i + deltaCounter)];
00395   noteAr[*tickCounter].ticks += tick;
00396   *i += deltaCounter;
00397 }
```

4.1.4.6 void countTicks2 (int *hex[]*, int * *i*, int *deltaCounter*, note *noteAr[]*, int * *tickCounter*)

Processes events with one parameter, extracting deltatime (and advancing the file pointer)

```
00401                                                                                                 {
00402   noteAr[*tickCounter].ticks = 0;
00403   int tick = 0;
00404
00405   while(deltaCounter < 6 && hex[(*i + deltaCounter)] > 0x80)
00406     tick += ((hex[(*i + deltaCounter++)] - 0x80) << 7);
00407
00408   tick += hex[(*i + deltaCounter)];
00409   noteAr[*tickCounter].ticks += tick;
00410   *i += deltaCounter;
00411 }
```

4.1.4.7 void deltaTimeToNoteLength (int *ppqn*, int *size*, note * *noteAr*)

Finds the note length, converted from deltatime to standard musical notation

```
00579                                                                                                 {
00580   for (int i = 0; i < size; i++){
00581     double noteLength = ((double) (noteAr[i].ticks)) / ((double) (ppqn/8));
00582
00583     if (noteLength < 1.5 && noteLength >= 0)
00584       noteLength = 1;
00585     else if (noteLength < 3 && noteLength >= 1.5)
00586       noteLength = 2;
00587     else if (noteLength < 6 && noteLength >= 3)
00588       noteLength = 4;
00589     else if (noteLength < 12 && noteLength >= 6)
00590       noteLength = 8;
00591     else if (noteLength < 24 && noteLength >= 12)
00592       noteLength = 16;
00593     else
00594       noteLength = 32;
00595
00596     noteAr[i].length = noteLength;
00597   }
00598 }
```

4.1.4.8 void fillNote (int *inputTone*, note * *note*)

A function to fill out each of the structures of type note

Parameters

inputTone	the value of the hexadecimal collected on the "tone"-spot
*note**	a pointer to a note-structure

```
00417                                                 {
00418   note->tone = inputTone % 12;
00419   note->average = inputTone;
00420   note->octave = inputTone / 12;
00421 }
```

4.1.4.9 void fillSongData (data * *data*, int *hex[]*, int *numbersInText*)

! A function, that fills out the song data

Parameters

**data*	a pointer to a structure containing the tempo and mode of the song
hex[]	the array of integers read from the file
numbersInText	the total amount of integers in the array

```
00270                                                                 {
00271   data->ppqn = (hex[12] << 8) + hex[13];
00272
00273   /*Find the mode of the song, initialised as minor atm*/
00274   for(int j = 0; j < numbersInText; j++)
```

```
00275     /* finds the tempo */
00276     if(hex[j] == 0xff && hex[j+1] == 0x51 && hex[j+2] == 0x03)
00277       data->tempo =  60000000/((hex[j+3] << 16) | (hex[j+4] << 8) | (hex[j+5]));
00278 }
```

4.1.4.10 void findEvents (int *numbersInText,* int *hex[],* **eventPlacement** *placement[],* **note** *noteAr[],* int * *size,* int * *amountOfNotes*)

Searches the file for events and stores their placement in an array of **eventPlacement** (p. 2) structs

```
00282
                {
00283   int noteOff = 0, noteOn = 0, afterTouch = 0, controlChange = 0,
00284       programChange = 0, channelPressure = 0, pitchWheel = 0, notes[numbersInText];
00285
00286   for(int j = 0; j < numbersInText; j++)
00287     switch (hex[j]){
00288       case 0x90: insertPlacement1(hex, &placement[noteOn++].noteOn, j, noteAr, amountOfNotes, notes);
              break;
00289       case 0x80: insertPlacement1(hex, &placement[noteOff++].noteOff, j, noteAr, amountOfNotes, notes);
              break;
00290       case 0xA0: insertPlacement1(hex, &placement[afterTouch++].afterTouch, j, noteAr, amountOfNotes, notes
      );       break;
00291       case 0xB0: insertPlacement1(hex, &placement[controlChange++].controlChange, j, noteAr, amountOfNotes,
       notes); break;
00292       case 0xC0: insertPlacement2(hex, &placement[programChange++].programChange, j);
              break;
00293       case 0xD0: insertPlacement2(hex, &placement[channelPressure++].channelPressure, j);
              break;
00294       case 0xE0: insertPlacement1(hex, &placement[pitchWheel++].pitchWheel, j, noteAr, amountOfNotes, notes
      );       break;
00295       default  :
              break;
00296     }
00297   findTicks(numbersInText, hex, placement, noteAr, noteOn, size, notes);
00298 }
```

4.1.4.11 void findMode (note * , int , data *)

4.1.4.12 void findMode (note *noteAr[],* int *totalNotes,* data * *data*)

A function to find the mode of the song by first calculating the tone span over sets of notes in the song, and then comparing it to the definition of minor and major keys.

Parameters

noteAr	An array of all the notes in the entire song
totalNotes	The number of notes in the song
data	The song data

```
00623                                                   {
00624   int majors[12] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, minors[12] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
00625   int x = 0, y = 0, z = 0, bar[4], sizeBar = 4, tempSpan = 999, span = 999, keynote = 0,
      mode = 0, tempNote = 0;
00626
00627   for(x = 0; x < totalNotes; x++){
00628     tempNote = noteAr[x].tone;
00629
00630     for(y = C; y <= B; y++)
00631       if(majors[y])
00632         checkScale(majors, tempNote, y);
00633   }
00634
00635   for(y = 0; y < 12; y++){
00636     z = y;
00637
00638     if(majors[z]){
00639       if((z - 3) < 0)
00640         z += 12;
00641
00642       minors[z-3] = 1;
00643     }
00644   }
00645
00646   z = 0;  x = 0;
00647
```

```
00648   /*Goes through all notes of the song and puts them into an array, 4 at a time*/
00649   while(x < totalNotes){
00650     z = x;
00651
00652     for(y = 0; y < sizeBar; y++, z++){
00653       if(z < totalNotes)
00654         bar[y] = noteAr[z].tone;
00655       else
00656         sizeBar = y;
00657     }
00658
00659     if(y == sizeBar){
00660       span = 999;
00661       /*Sort notes in ascending order*/
00662       qsort(bar, sizeBar, sizeof(tone), sortTones);
00663
00664       /*Finds the lowest possible tonespan over the array of 4 notes*/
00665       for(z = 0; z < sizeBar; z++){
00666           if((z + 1) > 3)
00667           tempSpan = (bar[(z+1)%4]+12)-bar[z] + bar[(z+2)%4]-bar[(z+1)%4] + bar[(z+3)%4]-bar[(z+2)%4];
00668         else if((z + 2) > 3)
00669           tempSpan = bar[(z+1)]-bar[z] + (bar[(z+2)%4]+12)-bar[(z+1)%4] + bar[(z+3)%4]-bar[(z+2)%4];
00670           else if((z +3) > 3)
00671           tempSpan = bar[(z+1)]-bar[z] + bar[(z+2)]-bar[(z+1)] + (bar[(z+3)%4]+12)-bar[z];
00672           else
00673           tempSpan = bar[(z+1)]-bar[z] + bar[(z+2)]-bar[(z+1)] + bar[(z+3)]-bar[(z+2)];
00674           if(tempSpan < span && (majors[bar[z]] || minors[bar[z]])){
00675           span = tempSpan;
00676           keynote = bar[z];
00677         }
00678       }
00679
00680       mode += isInScale(keynote, bar, sizeBar);
00681       x++;
00682     }
00683   }
00684
00685   /*outputs result directly to the data struct*/
00686   if(mode > 0)
00687     data->mode = major;
00688   else if(mode < 0)
00689     data->mode = minor;
00690 }
```

4.1.4.13 int FindMoodAmount (FILE * *moods*)

```
00753                                {
00754   int i = 1;
00755
00756   while(fgetc(moods) != EOF)
00757     if(fgetc(moods) == '\n')
00758       i++;
00759
00760   rewind(moods);
00761   return i;
00762 }
```

4.1.4.14 void findTicks (int *numbersInText*, int *hex[]*, eventPlacement *placement[]*, note *noteAr[]*, int *noteOn*, int * *size*, int *notes[]*)

```
00346
                    {
00347   int tickCounter = 0, deltaCounter1 = 3, deltaCounter2 = 2;
00348
00349   for(int j = 0; j < noteOn; j++){
00350     for(int i = placement[j].noteOn; i < numbersInText; i++){
00351       if(hex[i] == 0x80){
00352         if(hex[i + 1] == notes[j]){
00353           tickCounter++;
00354           break;
00355         }
00356         else
00357           countTicks1(hex, &i, deltaCounter1, noteAr, &tickCounter);
00358       }
00359       else if(hex[i] == 0xA0){
00360         if(hex[i + 1] == notes[j] && hex[i + 2] == 0x00){
00361           tickCounter++;
00362           break;
00363         }
00364         else
00365           countTicks1(hex, &i, deltaCounter1, noteAr, &tickCounter);
00366       }
00367       else if(hex[i] == 0xD0){
```

```
00368         if(hex[i + 1] == 0x00){
00369           tickCounter++;
00370           break;
00371         }
00372         else
00373           countTicks2(hex, &i, deltaCounter2, noteAr, &tickCounter);
00374       }
00375       else if(hex[i] == 0xC0)
00376         countTicks2(hex, &i, deltaCounter2, noteAr, &tickCounter);
00377       else
00378         countTicks1(hex, &i, deltaCounter1, noteAr, &tickCounter);
00379     }
00380   }
00381
00382   *size = tickCounter;
00383 }
```

4.1.4.15 int getHex (FILE * *f*, int *hexAr[]*)

A function, that retrieves the hexadecimals from the files and also returns the number of files

Parameters

**f*	a pointer to the file the program is reading from
hexAr[]	an array of integers, that the information is stored in

```
00237                                   {
00238   int i = 0, c;
00239
00240   while( (c = fgetc(f)) != EOF && i < CHARS){
00241     hexAr[i] = c;
00242     i++;
00243   }
00244
00245   return i;
00246 }
```

4.1.4.16 void insertMoods (moodWeighting *moodArray[]*, FILE * *moods*)

Inserts the weighting of each mood in an array of structs, as read from a designated file.

Parameters

moodArray	The array moods are stored in
moods	the file to be read

```
00550                                                           {
00551   for(int i = 0; i < AMOUNT_OF_MOODS; i++)
00552     fscanf(moods, "%s %d %d %d %d", moodArray[i].name , &moodArray[i].mode,
00553                                     &moodArray[i].tempo, &moodArray[i].toneLength,
00554                                     &moodArray[i].pitch);
00555 }
```

4.1.4.17 void insertPlacement1 (int *hex[]*, int * *place*, int *j*, note *noteAr[]*, int * *amountOfNotes*, int *notes[]*)

Starts in the hex which are investigated and looks forward to find a perspective. It goes to an assumed deltatime and finds the length of it. Thereafter it checks the next hex after the deltatime to make sure it is an event. If that is the case it stores the hex which is investegated in the first place. Furthermore if it is a noteOn event it stores the hex which is the note, processes the note and counts amount of notes.

```
00305                                                                                   {
00306   int i = 3;
00307
00308   while(i < 7 && hex[(j + i++)] > 0x80);
00309
00310   if(checkNextEvent(hex, (j + i))){
00311     *place = j;
00312     if(hex[j] == 0x90){
00313       notes[*amountOfNotes] = hex[j + 1];
00314       fillNote(hex[j + 1], &noteAr[*amountOfNotes]);
00315       *amountOfNotes += 1;
00316     }
00317   }
00318 }
```

4.1.4.18 void insertPlacement2 (int *hex[]*, int * *place*, int *j*)

Does the same as insertPlacement1, but for events with 1 parameter.

```
00322                                                   {
00323   int i = 2;
00324
00325   while(i < 6 && hex[(j + i++)] > 0x80);
00326
00327   if(checkNextEvent(hex, (j + i)))
00328     *place = j;
00329 }
```

4.1.4.19 int isInMajor (int *toneLeap*)

A function to check if the given tone leap is in the major scale.

Parameters

toneLeap	An integer describing the processed tone leap

Returns

a boolean value, returns 1 if the tone leap is in the major scale, 0 if it's not.

```
00742                           {
00743   int major[] = {0, 2, 4, 5, 7, 9, 11};
00744
00745   for(int i = 0; i < SCALESIZE; i++)
00746     if(toneLeap == major[i])
00747       return 1;
00748
00749   return 0;
00750 }
```

4.1.4.20 int isInMinor (int *toneLeap*)

A function to check if the given tone leap is in the minor scale.

Parameters

toneLeap	An integer describing the processed tone leap

Returns

a boolean value, returns 1 if the tone leap is in the minor scale, 0 if it's not.

```
00728                           {
00729   int minor[] = {0, 2, 3, 5, 7, 8, 10};
00730
00731   for(int i = 0; i < SCALESIZE; i++)
00732     if(toneLeap == minor[i])
00733       return 1;
00734
00735   return 0;
00736 }
```

4.1.4.21 int isInScale (int *keytone*, int *otherTones[]*, int *size*)

A function to check if a given scale in given keytone corresponds with the tones in the rest of the song.

Parameters

keytone	The keytone of the processed scale
otherTones	An array of the rest of the tones, which the function compares to the keytone and mode
size	The number of tones in the otherTones array

Returns

a boolean value, returns 1 if the mode is major, -1 if it's minor and 0, if wasn't possible to decide.

```
00698                                                          {
00699   int toneLeap, isMinor = 1, isMajor = 1;
00700
00701   for(int i = 0; i < size; i++){
00702     if(otherTones[i] < keytone)
00703       otherTones[i] += 12;
00704
00705     toneLeap = otherTones[i] - keytone;
00706
00707     if(isMinor)
00708       isMinor = isInMinor(toneLeap);
00709
00710     if(isMajor)
00711       isMajor = isInMajor(toneLeap);
00712   }
00713
00714   if(isMinor && isMajor)
00715     return 0;
00716   else if(isMinor)
00717     return -1;
00718   else if(isMajor)
00719     return 1;
00720
00721   return 0;
00722 }
```

4.1.4.22 int main (int *argc*, const char * *argv[]*)

```
00116                                       {
00117   DIR *dir = 0;
00118   FILE *f;
00119   char MIDIfile[25];
00120   /*Variables*/
00121   int numbersInText = 0, notes, size = 0, mode = 5, tempo = 5, toneLength = 5, pitch = 5, amountOfNotes = 0
      ;
00122   FILE* moods = fopen("moods.txt", "r");
00123
00124   if(moods == NULL){
00125     perror("Error: moods missing ");
00126     exit(EXIT_FAILURE);
00127   }
00128
00129   AMOUNT_OF_MOODS = FindMoodAmount(moods);
00130   moodWeighting moodArray[AMOUNT_OF_MOODS];
00131   data data = {0, major, D};
00132
00133   if (argv[1] == NULL){
00134     checkDirectory(MIDIfile, dir);
00135     f = fopen(MIDIfile, "r");
00136
00137     if(f == NULL){
00138       perror("Error opening file");
00139       exit(EXIT_FAILURE);
00140     }
00141   }
00142   else if(argv[1] != NULL){
00143     f = fopen(argv[1],"r");
00144
00145     if(f == NULL){
00146       perror("Error opening file");
00147       exit(EXIT_FAILURE);
00148     }
00149   }
00150
00151   closedir (dir);
00152   int *hex = (int *) malloc(CHARS * sizeof(int));
00153
00154   if(hex == NULL){
00155     printf("Memory allocation failed, bye!");
00156     exit(EXIT_FAILURE);
00157   }
00158
00159   /*Reading the data from the file*/
```

```
00160   numbersInText = getHex(f, hex);
00161   fillSongData(&data, hex, numbersInText);
00162   notes = countPotentialNotes(hex, numbersInText);
00163   note *noteAr = (note*) malloc(notes * sizeof(note));
00164
00165   if(noteAr == NULL){
00166     printf("Memory allocation failed, bye!");
00167     exit(EXIT_FAILURE);
00168   }
00169
00170   eventPlacement placement[numbersInText];
00171   int result[AMOUNT_OF_MOODS];
00172   findEvents(numbersInText, hex, placement, noteAr, &size, &amountOfNotes);
00173   deltaTimeToNoteLength(data.ppqn, size, noteAr);
00174   insertMoods(moodArray, moods);
00175   findMode(noteAr, amountOfNotes, &data);
00176   settingPoints(&mode, &tempo, &toneLength, &pitch, data, amountOfNotes, noteAr, &size);
00177   weightingMatrix(moodArray, mode, tempo, toneLength, pitch, result);
00178
00179   /*Clean up and close*/
00180   fclose(f);
00181   free(hex);
00182   free(noteAr);
00183
00184   /* Print results */
00185   printResults(mode, tempo, toneLength, pitch, moodArray, result);
00186
00187   return 0;
00188 }
```

4.1.4.23 void printNote (note *note*)

A function to print the note

Parameters

note	the note structure to be printed

```
00426                         {
00427   printf("Tone: ");
00428
00429   switch (note.tone){
00430     case C     : printf("C") ; break;
00431     case Csharp: printf("C#"); break;
00432     case D     : printf("D") ; break;
00433     case Dsharp: printf("D#"); break;
00434     case E     : printf("E") ; break;
00435     case F     : printf("F") ; break;
00436     case Fsharp: printf("F#"); break;
00437     case G     : printf("G") ; break;
00438     case Gsharp: printf("G#"); break;
00439     case A     : printf("A") ; break;
00440     case Asharp: printf("A#"); break;
00441     case B     : printf("B") ; break;
00442     default    : printf("Undefined note"); break;
00443   }
00444
00445   printf(", octave: %d\n", note.octave);
00446 }
```

4.1.4.24 void printResults (int *mode,* int *tempo,* int *toneLength,* int *pitch,* moodWeighting *moodArray[],* int *result[]*)

Prints relevant information about the song. Finds and prints the mood with the highest score, and in the case of using the default sad/happy scale, scales the values to fit on the 51 point sliding scale

```
00768                                                                                                                {
00769   printf("\n\n\n");
00770   printf(" Mode:");
00771   printf("%10d\n", mode);
00772   printf(" Tempo:");
00773   printf("%9d\n", tempo);
00774   printf(" Tone length:");
00775   printf("%3d\n", toneLength);
00776   printf(" Pitch:");
00777   printf("%9d\n", pitch);
00778   printf("\n\n\n                                                  WEIGHTINGS\n");
00779   printf("                           Mode | Tempo | Tone length | Pitch\n");
00780
```

```
00781   for(int i = 0; i < AMOUNT_OF_MOODS; i++){
00782     printf(" %s", moodArray[i].name);
00783     for(int j = strlen(moodArray[i].name); j < 26; j++)
00784       printf(" ");
00785     if(moodArray[i].mode > -1)
00786       printf(" ");
00787     printf(" %d", moodArray[i].mode);
00788     for(int j = 0; j < 2; j++)
00789       printf(" ");
00790     printf("| ");
00791     if(moodArray[i].tempo > -1)
00792       printf(" ");
00793     printf(" %d", moodArray[i].tempo);
00794     for(int j = 0; j < 3; j++)
00795       printf(" ");
00796     printf("|    ");
00797     if(moodArray[i].toneLength > -1)
00798       printf(" ");
00799     printf(" %d", moodArray[i].toneLength);
00800     for(int j = 0; j < 6; j++)
00801       printf(" ");
00802     printf("|  ");
00803     if(moodArray[i].pitch > -1)
00804       printf(" ");
00805     printf(" %d\n", moodArray[i].pitch);
00806   }
00807
00808   printf("\n\n\n");
00809
00810   for(int i = 0; i < AMOUNT_OF_MOODS; i++){
00811     if(mode < 0)
00812       printf(" %d * ", mode);
00813     else
00814       printf(" %d * ", mode);
00815     if(moodArray[i].mode < 0)
00816       printf("%d + ", moodArray[i].mode);
00817     else
00818       printf(" %d + ", moodArray[i].mode);
00819     if(tempo < 0)
00820       printf("%d * ", tempo);
00821     else
00822       printf(" %d * ", tempo);
00823     if(moodArray[i].tempo < 0)
00824       printf("%d + ", moodArray[i].tempo);
00825     else
00826       printf(" %d + ", moodArray[i].tempo);
00827     if(toneLength < 0)
00828       printf("%d * ", toneLength);
00829     else
00830       printf(" %d * ", toneLength);
00831     if(moodArray[i].toneLength < 0)
00832       printf("%d + ", moodArray[i].toneLength);
00833     else
00834       printf(" %d + ", moodArray[i].toneLength);
00835     if(pitch < 0)
00836       printf("%d * ", pitch);
00837     else
00838       printf(" %d * ", pitch);
00839     if(moodArray[i].pitch < 0)
00840       printf("%d = ", moodArray[i].pitch);
00841     else
00842       printf(" %d = ", moodArray[i].pitch);
00843     if(result[i] < 0)
00844       printf("%d\n", result[i]);
00845     else
00846       printf(" %d\n", result[i]);
00847   }
00848
00849   int moodOfMelodi = 0, test = 0;
00850
00851   for(int i = 0; i < AMOUNT_OF_MOODS; i++)
00852     if(moodOfMelodi < result[i])
00853       moodOfMelodi = i;
00854
00855   if(!strcmp(moodArray[moodOfMelodi].name, "Happy")){
00856     printf("\n\n\n Sad ");
00857
00858     while(test < 51){
00859       if(test == 25)
00860         printf("|");
00861       else if(test == ((result[moodOfMelodi] / 2) + 26))
00862         printf("[]");
00863       else
00864         printf("-");
00865
00866       test++;
00867     }
```

```
00868
00869     printf(" Happy\n\n\n");
00870   }
00871   else if(!strcmp(moodArray[moodOfMelodi].name, "Sad")){
00872     printf("\n\n\n Sad ");
00873
00874     while(test < 51){
00875       if(test == 25)
00876         printf("|");
00877       else if(test == ((int)(-((result[moodOfMelodi]) / 2.4)) + 26))
00878         printf("[]");
00879       else
00880         printf("-");
00881
00882       test++;
00883     }
00884
00885     printf(" Happy\n\n\n");
00886   }
00887
00888   printf("\n The mood of the melody is %s\n", moodArray[moodOfMelodi].name);
00889 }
```

4.1.4.25 void settingPoints (int * *mode*, int * *tempo*, int * *length*, int * *octave*, data *data*, int *notes*, note *noteAr[]*, int * *size*)

A function to insert points into integers based on the data pulled from the file

Parameters

mode,along	with tempo, length and octave contains the points
data	contains the song data
notes	contains the amount of notes in the song
note	contains an array of the specific notes

```
00454
          {
00455   int deltaTime = 0, combined = 0, averageNote = 0;
00456
00457   switch(data.mode){
00458     case minor: *mode = -5; break;
00459     case major: *mode = 5; break;
00460     default: *mode = 0; break;
00461   }
00462
00463   if(data.tempo < 60)
00464     *tempo = -5;
00465   else if(data.tempo >= 60 && data.tempo < 70)
00466     *tempo = -4;
00467   else if(data.tempo >= 70 && data.tempo < 80)
00468     *tempo = -3;
00469   else if(data.tempo >= 80 && data.tempo < 90)
00470     *tempo = -2;
00471   else if(data.tempo >= 90 && data.tempo < 100)
00472     *tempo = -1;
00473   else if(data.tempo >= 100 && data.tempo < 120)
00474     *tempo =  0;
00475   else if(data.tempo >= 120 && data.tempo < 130)
00476     *tempo =  1;
00477   else if(data.tempo >= 130 && data.tempo < 140)
00478     *tempo =  2;
00479   else if(data.tempo >= 140 && data.tempo < 150)
00480     *tempo =  3;
00481   else if(data.tempo >= 150 && data.tempo < 160)
00482     *tempo =  4;
00483   else if(data.tempo >=  160)
00484     *tempo =  5;
00485
00486   for(int i = 0; i < notes; i++)
00487     combined += noteAr[i].length;
00488
00489   deltaTime = combined/notes;
00490
00491   if (deltaTime < 1.5 && deltaTime >= 0)
00492     *length = 5;
00493   else if (deltaTime < 3 && deltaTime >= 1.5)
00494     *length = 4;
00495   else if (deltaTime < 5 && deltaTime >= 4)
00496     *length = 3;
00497   else if (deltaTime < 6 && deltaTime >= 5)
00498     *length = 2;
```

```
00499   else if (deltaTime < 9 && deltaTime >= 6)
00500     *length = 1;
00501   else if (deltaTime < 12 && deltaTime >= 9)
00502     *length = 0;
00503   else if (deltaTime < 16 && deltaTime >= 12)
00504     *length = -1;
00505   else if (deltaTime < 20 && deltaTime >= 16)
00506     *length = -2;
00507   else if (deltaTime < 24 && deltaTime >= 20)
00508     *length = -3;
00509   else if (deltaTime < 28 && deltaTime >= 24)
00510     *length = -4;
00511   else
00512     *length = -5;
00513
00514   combined = 0;
00515
00516   for (int i = 0; i < notes; i++)
00517     combined += noteAr[i].average;
00518
00519   averageNote = combined/notes;
00520
00521   if(averageNote <= 16)
00522     *octave = -5;
00523   else if(averageNote >= 17 && averageNote <= 23)
00524     *octave = -4;
00525   else if(averageNote >= 24 && averageNote <= 30)
00526     *octave = -3;
00527   else if(averageNote >= 31 && averageNote <= 37)
00528     *octave = -2;
00529   else if(averageNote >= 38 && averageNote <= 44)
00530     *octave = -1;
00531   else if(averageNote >= 45 && averageNote <= 51)
00532     *octave = 0;
00533   else if(averageNote >= 52 && averageNote <= 58)
00534     *octave = 1;
00535   else if(averageNote >= 59 && averageNote <= 65)
00536     *octave = 2;
00537   else if(averageNote >= 66 && averageNote <= 72)
00538     *octave = 3;
00539   else if(averageNote >= 73 && averageNote <= 79)
00540     *octave = 4;
00541   else if(averageNote >=80)
00542     *octave = 5;
00543 }
```

4.1.4.26 int sortToner (const void *, const void *)

4.1.4.27 int sortTones (const void * *a*, const void * *b*)

A function to sort integers in ascending order, used by qsort

```
00602                                           {
00603   return (*(int *)a - *(int *)b);
00604 }
```

4.1.4.28 void weightingMatrix (moodWeighting *moodArray[]*, int *mode*, int *tempo*, int *toneLength*, int *pitch*, int * *result*)

Vector matrix multiplication. Receives an array of moods, the various parameters of the song and a pointer to an array where the results will be stored. The song data is multiplied onto each moods weighting and then stored.

Parameters

moodArray	an array containing the weighting for all moods
result	an array for holding the songs scores as per each mood
mode	along with temp, toneLength and pitch, this variable contains a score -5 to 5 for how that facet of the song is.

```
00565
       {
00566   for(int i = 0; i < AMOUNT_OF_MOODS; i++)
00567     result[i] = 0;
00568
00569   for(int i = 0; i < AMOUNT_OF_MOODS; i++){
00570     result[i] += (moodArray[i].mode * mode);
00571     result[i] += (moodArray[i].tempo * tempo);
```

```
00572     result[i] += (moodArray[i].toneLength * toneLength);
00573     result[i] += (moodArray[i].pitch * pitch);
00574   }
00575 }
```

4.1.5 Variable Documentation

4.1.5.1 int AMOUNT_OF_MOODS

Index