A412

0.1

Generated by Doxygen 1.8.8

Thu Dec 11 2014 14:58:14

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**
- **mode mode**
- **tone key**

3.1.1 Field Documentation

3.1.1.1 tone data::key

3.1.1.2 mode data::mode

3.1.1.3 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 Field Documentation

3.2.1.1 int eventPlacement::afterTouch

3.2.1.2 int eventPlacement::channelPressure

3.2.1.3 int eventPlacement::controlChange

3.2.1.4 int eventPlacement::noteOff

3.2.1.5 int eventPlacement::noteOn

3.2.1.6 int eventPlacement::pitchWheel

3.2.1.7 int eventPlacement::programChange

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

- **findEvents.c**
- **main.c**

3.3 moodWeighting Struct Reference

Data Fields

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

3.3.1 Field Documentation

3.3.1.1 int moodWeighting::mode

3.3.1.2 char moodWeighting::name[25]

3.3.1.3 int moodWeighting::pitch

3.3.1.4 int moodWeighting::tempo

3.3.1.5 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**

3.4.1 Field Documentation

3.4.1.1 int note::average

3.4.1.2 int note::length

3.4.1.3 int note::octave

3.4.1.4 int note::tone

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

- **main.c**

3.5 points Struct Reference

Data Fields

- char * **parameter**
- int **point**

3.5.1 Field Documentation

3.5.1.1 char* points::parameter

3.5.1.2 int points::point

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

- **main.c**

4 File Documentation

4.1 findEvents.c File Reference

Data Structures

- struct **eventPlacement**

Functions

- int **main** (void)
- void **findEvents** (int numbersInText, int hex[], **eventPlacement** placement[], **note** noteAr[], int ticks[])
- void **insertPlacement1** (int hex[], int *place, int j, **note** noteAr[], int *n)
- void **insertPlacement2** (int hex[], int *place, int j)
- int **checkNextEvent** (int hex[], int j)
- void **findTicks** (int numbersInText, int hex[], **eventPlacement** placement[], **note** noteAr[], int ticks[])
- void **countTicks1** (int hex[], int *i, int deltaCounter, int ticks[], int *tickCounter)
- void **countTicks2** (int hex[], int *i, int deltaCounter, int ticks[], int *tickCounter)

4.1.1 Function Documentation

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

```
{
  switch (hex[j]){
    case 0x90:
    case 0x80:
    case 0xA0:
    case 0xB0:
    case 0xC0:
    case 0xD0:
    case 0xE0: return 1; break;
    default  : return 0; break;
  }
}
```

4.1.1.2 void countTicks1 (int *hex[ ],* int ∗ *i,* int *deltaCounter,* int *ticks[ ],* int ∗ *tickCounter*)

```
{
  while(deltaCounter < 7 && hex[(i + deltaCounter)] > 0x80)
    ticks[tickCounter] += ((hex[(i + deltaCounter++)] - 0x80) * 128);
  ticks[tickCounter++] += hex[(i + deltaCounter++)];
  i += deltaCounter;
}
```

4.1.1.3 void countTicks2 (int *hex[ ],* int ∗ *i,* int *deltaCounter,* int *ticks[ ],* int ∗ *tickCounter*)

```
{
  while(deltaCounter < 6 && hex[(i + deltaCounter)] > 0x80)
    ticks[tickCounter] += ((hex[(i + deltaCounter++)] - 0x80) * 128);
  ticks[tickCounter++] += hex[(i + deltaCounter++)];
  i += deltaCounter;
}
```

4.1.1.4 void findEvents (int *numbersInText,* int *hex[ ],* eventPlacement *placement[ ],* note *noteAr[ ],* int *ticks[ ]*)

```
{
  int noteOff = 0, noteOn = 0, afterTouch = 0, controlChange = 0,
      programChange = 0, channelPressure = 0, pitchWheel = 0, i = 0, n = 0;

  for(int j = 0; j < numbersInText; j++){
    switch (hex[j]){
      case 0x90: insertPlacement1(hex, &placement[noteOn++].noteOn, j, noteAr, &n);             break;
      case 0x80: insertPlacement1(hex, &placement[noteOff++].noteOff, j, noteAr, &n);           break;
      case 0xA0: insertPlacement1(hex, &placement[afterTouch++].afterTouch, j, noteAr, &n);     break;
      case 0xB0: insertPlacement1(hex, &placement[controlChange++].controlChange, j, noteAr, &n); break;
      case 0xC0: insertPlacement2(hex, &placement[programChange++].programChange, j);           break;
      case 0xD0: insertPlacement2(hex, &placement[channelPressure++].channelPressure, j);       break;
      case 0xE0: insertPlacement1(hex, &placement[pitchWheel++].pitchWheel, j, noteAr, &n);     break;
      default  :                                                                                 break;
    }
  }
  findTicks(numbersInText, hex, placement, noteAr, ticks);
}
```

4.1.1.5 void findTicks (int *numbersInText,* int *hex[ ],* eventPlacement *placement[ ],* note *noteAr[ ],* int *ticks[ ]*)

```
{
  int tickCounter = 0, deltaCounter1 = 3, deltaCounter2 = 2;

  for(int j = 0; j < noteOn; j++){
    for(int i = placement[j].noteOn; i < numbersInText; i++){
      if(hex[i] == 0x80){
        if(hex[i + 1] == noteAr[j])
          break;
        else{
          countTicks1(hex, &i, deltaCounter1, ticks[], tickCounter);
        }
      }
      else if(hex[i] == 0xA0){
        if(hex[i + 1] == noteAr[j] && hex[i + 2] == 0x00)
          break;
        else{
          countTicks1(hex, &i, deltaCounter1, ticks[], tickCounter);
        }
      }
```

```
00086       else if(hex[i] == 0xD0){
00087         if(hex[i + 1] == 0x00)
00088           break;
00089         else{
00090           countTicks2(hex, &i, deltaCounter2, ticks[], tickCounter);
00091         }
00092       }
00093       else if(hex[start] == 0xC0){
00094         countTicks2(hex, &i, deltaCounter2, ticks[], tickCounter);
00095       }
00096       else{
00097         countTicks1(hex, &i, deltaCounter1, ticks[], tickCounter);
00098       }
00099     }
00100   }
00101 }
```

4.1.1.6 void insertPlacement1 (int *hex[]*, int * *place*, int *j*, note *noteAr[]*, int * *n*)

```
00035                                                                    {
00036   int i = 3;
00037   while(i < 7 && hex[(j + i++)] > 0x80);
00038   if(checkNextEvent(hex, (j + i))){
00039     *place = j;
00040     if(hex[j] == 0x90){
00041       fillNote(hex[j + 1], &noteAr[*n]);
00042       *n += 1;
00043     }
00044   }
00045 }
```

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

```
00047                                                 {
00048   int i = 2;
00049   while(i < 6 && hex[(j + i++)] > 0x80);
00050   if(checkNextEvent(hex, (j + i)))
00051     *place = j;
00052 }
```

4.1.1.8 int main (void)

```
00011               {
00012   int ticks[numbersInText];
00013   return 0;
00014 }
```

4.2 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 **points**
- 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 **findNoteLength** (double x, int ∗, int ∗)
- void **printNote** (**note**)
- int **getHex** (FILE ∗, int[])
- void **fillSongData** (**data** ∗, int[], int)
- int **countNotes** (int[], int)
- void **fillNote** (int, **note** ∗)
- void **printSongData** (**data**)
- void **settingPoints** (int ∗, int ∗, int ∗, int ∗, **data**, int, **note**[], int ∗)
- void **insertMoods** (**moodWeighting**[], FILE ∗)
- int **weightingMatrix** (**moodWeighting**[], int, int, int, int)
- void **findEvents** (int, int[], **eventPlacement**[], **note**[], int[], int ∗)
- void **insertPlacement1** (int[], int ∗, int, **note**[], int ∗)
- void **insertPlacement2** (int[], int ∗, int)
- int **checkNextEvent** (int[], int)
- void **findTicks** (int, int[], **eventPlacement**[], **note**[], int[], int, int ∗)
- void **countTicks1** (int[], int ∗, int, int[], int ∗)
- void **countTicks2** (int[], int ∗, int, int[], int ∗)
- int **sortResult** (const void ∗, const void ∗)
- void **deltaTimeToNoteLength** (int ∗, 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 ∗)
- 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**

4.2.1 Macro Definition Documentation

4.2.1.1 #define CHARS 1000

4.2.1.2 #define SCALESIZE 7

4.2.2 Typedef Documentation

4.2.2.1 typedef enum mode mode

4.2.2.2 typedef enum mood mood

4.2.2.3 typedef enum tone tone

4.2.3 Enumeration Type Documentation

4.2.3.1 enum mode

Enumerator

major

minor

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

4.2.3.2 enum mood

Enumerator

glad

sad

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

4.2.3.3 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;
```

4.2.4 Function Documentation

4.2.4.1 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

```
00164                                           {
00165   struct dirent *musicDir;
00166   if ((dir = opendir ("./Music")) != NULL) {
00167     printf("Mulige numre\n");
00168       while ((musicDir = readdir (dir)) != NULL) {
00169         printf ("%s\n", musicDir->d_name);
00170       }
00171   }
00172   else {
00173     perror ("Failure while opening directory");
00174     exit (EXIT_FAILURE);
00175   }
00176   printf("Indtast det valgte nummer\n");
00177   scanf("%s", MIDIfile);
00178 }
```

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

```
00263                                       {
00264   switch (hex[j]){
00265     case 0x90:
00266     case 0x80:
00267     case 0xA0:
00268     case 0xB0:
00269     case 0xC0:
00270     case 0xD0:
00271     case 0xE0: return 1; break;
00272     default  : return 0; break;
00273   }
00274 }
```

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

```
00513                                                   {
00514   if(tone < key)
00515     tone += 12;
00516   scales[key] = isInMajor(tone - key);
00517 }
```

4.2.4.4 int countNotes (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

```
00198                                       {
00199   int i = 0, res = 0;
00200   for(i = 0; i < amount; i++){
00201     if(hex[i] == 0x90){
00202       res++;
00203     }
00204   }
00205   return res;
00206 }
```

4.2.4.5 void countTicks1 (int *hex[]*, int * *i*, int *deltaCounter*, int *ticks[]*, int * *tickCounter*)

```
00312                                                                                                   {
00313   while(deltaCounter < 7 && hex[(*i + deltaCounter)] > 0x80)
00314     ticks[*tickCounter] += ((hex[(*i + deltaCounter++)] - 0x80) * 128);
00315   ticks[*tickCounter++] += hex[(*i + deltaCounter++)];
00316   i += deltaCounter;
00317 }
```

4.2.4.6 void countTicks2 (int *hex[],* int * *i,* int *deltaCounter,* int *ticks[],* int * *tickCounter*)

```
00319                                                                                                  {
00320   while(deltaCounter < 6 && hex[(*i + deltaCounter)] > 0x80)
00321     ticks[*tickCounter] += ((hex[(*i + deltaCounter++)] - 0x80) * 128);
00322   ticks[*tickCounter++] += hex[(*i + deltaCounter++)];
00323   i += deltaCounter;
00324 }
```

4.2.4.7 void deltaTimeToNoteLength (int * *ticks,* int *ppqn,* int *size,* note * *noteAr*)

```
00482                                                                                                  {
00483
00484   for (int i = 0; i < size; i++){
00485
00486     double noteLength = ((double) (ticks[i])) / ((double) (ppqn/8));
00487
00488     if (noteLength < 1.5 && noteLength >= 0)
00489       noteLength = 1;
00490     else if (noteLength < 3 && noteLength >= 1.5)
00491       noteLength = 2;
00492     else if (noteLength < 6 && noteLength >= 3)
00493       noteLength = 4;
00494     else if (noteLength < 12 && noteLength >= 6)
00495       noteLength = 8;
00496     else if (noteLength < 24 && noteLength >= 12)
00497       noteLength = 16;
00498     else
00499       noteLength = 32;
00500
00501         noteAr[i].length = noteLength;
00502     }
00503 }
```

4.2.4.8 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

```
00330                                          {
00331   note->tone = inputTone % 12;
00332   note->average = inputTone;
00333   note->octave = inputTone / 12;
00334 }
```

4.2.4.9 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

```
00214                                                          {
00215   int j;
00216   /*Find the mode of the song, initialised as minor atm*/
00217   for(j = 0; j < numbersInText; j++){
00218     /* finds the tempo */
00219     if(hex[j] == 0xff && hex[j+1] == 0x51 && hex[j+2] == 0x03){
00220       data->tempo =  60000000/((hex[j+3] << 16) | (hex[j+4] << 8) | (hex[j+5]));
00221     }
00222   }
00223 }
```

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

```
00225
        {
00226   int noteOff = 0, noteOn = 0, afterTouch = 0, controlChange = 0,
00227       programChange = 0, channelPressure = 0, pitchWheel = 0, n = 0;
00228
00229   for(int j = 0; j < numbersInText; j++){
00230     switch (hex[j]){
00231       case 0x90: insertPlacement1(hex, &placement[noteOn++].noteOn, j, noteAr, &n);               break;
00232       case 0x80: insertPlacement1(hex, &placement[noteOff++].noteOff, j, noteAr, &n);             break;
00233       case 0xA0: insertPlacement1(hex, &placement[afterTouch++].afterTouch, j, noteAr, &n);       break;
00234       case 0xB0: insertPlacement1(hex, &placement[controlChange++].controlChange, j, noteAr, &n); break;
00235       case 0xC0: insertPlacement2(hex, &placement[programChange++].programChange, j);             break;
00236       case 0xD0: insertPlacement2(hex, &placement[channelPressure++].channelPressure, j);         break;
00237       case 0xE0: insertPlacement1(hex, &placement[pitchWheel++].pitchWheel, j, noteAr, &n);       break;
00238       default  :                                                                                  break;
00239     }
00240   }
00241   findTicks(numbersInText, hex, placement, noteAr, ticks, noteOn, size);
00242 }
```

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

4.2.4.12 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

```
00524                                                   {
00525   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};
00526   int x = 0, y = 0, z = 0, bar[4], sizeBar = 4, tempSpan = 999, span = 999, keynote = 0,
      mode = 0, tempNote = 0;
00527
00528   for(x = 0; x < totalNotes; x++){
00529     tempNote = noteAr[x].tone;
00530
00531     for(y = C; y <= B; y++){
00532       if(majors[y])
00533         checkScale(majors, tempNote, y);
00534     }
00535   }
00536
00537   /*TEST Keynote giver det forkerte svar, resten virker*/
00538   for(y = 0; y < 12; y++){
00539     z = y;
00540     if(majors[z]){
00541       if((z - 3) < 0)
00542         z += 12;
00543     minors[z-3] = 1;
00544     }
00545   }
00546   for(int p = 0; p < 12; p++){
00547     printf("Dur: %d\n", majors[p]);
00548   }
00549   for(int p = 0; p < 12; p++)
00550     printf("Mol: %d\n", minors[p]);
00551   z = 0;
00552   x = 0;
00553
00554   /*Goes through all notes of the song and puts them into an array*/
00555   while(x < totalNotes){
00556     z = x;
00557     for(y = 0; y < sizeBar; y++, z++){
00558       if(z < totalNotes)
00559         bar[y] = noteAr[z].tone;
00560       else
00561         sizeBar = y;
00562     }
00563
00564     if(y == sizeBar){
00565       span = 999;
00566       /*Sort notes in acsending order*/
00567       qsort(bar, sizeBar, sizeof(tone), sortTones);
```

```
00568
00569       /*Find the lowest possible tonespan over the entire array of notes*/
00570       for(z = 0; z < sizeBar; z++){
00571     if((z + 1) > 3)
00572           tempSpan = (bar[(z+1)%4]+12)-bar[z] + bar[(z+2)%4]-bar[(z+1)%4] + bar[(z+3)%4]-bar[(z+2)%4];
00573         else if((z + 2) > 3)
00574           tempSpan = bar[(z+1)]-bar[z] + (bar[(z+2)%4]+12)-bar[(z+1)%4] + bar[(z+3)%4]-bar[(z+2)%4];
00575     else if((z +3) > 3)
00576           tempSpan = bar[(z+1)]-bar[z] + bar[(z+2)]-bar[(z+1)] + (bar[(z+3)%4]+12)-bar[z];
00577     else
00578           tempSpan = bar[(z+1)]-bar[z] + bar[(z+2)]-bar[(z+1)] + bar[(z+3)]-bar[(z+2)];
00579
00580     if(tempSpan < span && (majors[bar[z]] || minors[bar[z]])){
00581           span = tempSpan;
00582           keynote = bar[z];
00583         }
00584       }
00585       mode += isInScale(keynote, bar, sizeBar);
00586       printf("Moden er nu: %d\n", mode);
00587       x++;
00588     }
00589   }
00590
00591   if(mode > 0)
00592     data->mode = major;
00593   else if(mode < 0)
00594     data->mode = minor;
00595 }
```

4.2.4.13 int FindMoodAmount (FILE * *moods*)

```
00657                                 {
00658   int i = 1;
00659   while(fgetc(moods) != EOF){
00660     if(fgetc(moods) == '\n')
00661       i++;
00662   }
00663   rewind(moods);
00664   return i;
00665 }
```

4.2.4.14 void findNoteLength (double *x*, int * , int *)

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

```
00276
                    {
00277   int tickCounter = 0, deltaCounter1 = 3, deltaCounter2 = 2;
00278
00279   for(int j = 0; j < noteOn; j++){
00280     for(int i = placement[j].noteOn; i < numbersInText; i++){
00281       if(hex[i] == 0x80){
00282         if(hex[i + 1] == noteAr[j].tone)
00283           break;
00284         else{
00285           countTicks1(hex, &i, deltaCounter1, ticks, &tickCounter);
00286         }
00287       }
00288       else if(hex[i] == 0xA0){
00289         if(hex[i + 1] == noteAr[j].tone && hex[i + 2] == 0x00)
00290           break;
00291         else{
00292           countTicks1(hex, &i, deltaCounter1, ticks, &tickCounter);
00293         }
00294       }
00295       else if(hex[i] == 0xD0){
00296         if(hex[i + 1] == 0x00)
00297           break;
00298         else{
00299           countTicks2(hex, &i, deltaCounter2, ticks, &tickCounter);
00300         }
00301       }
00302       else if(hex[i] == 0xC0){
00303         countTicks2(hex, &i, deltaCounter2, ticks, &tickCounter);
00304       }
00305       else{
00306         countTicks1(hex, &i, deltaCounter1, ticks, &tickCounter);
00307       }
00308     }
00309   }
00310 }
```

4.2.4.16 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

```
00184                                {
00185   int i = 0, c;
00186   while( (c = fgetc(f)) != EOF && i < CHARS){
00187     hexAr[i] = c;
00188     i++;
00189   }
00190
00191   return i;
00192 }
```

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

```
00443                                                          {
00444   for(int i = 0; i < AMOUNT_OF_MOODS; i++){
00445     fscanf(moods, "%s %d %d %d %d", moodArray[i].name , &moodArray[i].mode,
00446                                     &moodArray[i].tempo, &moodArray[i].toneLength,
00447                                     &moodArray[i].pitch);
00448   }
00449 }
```

4.2.4.18 void insertPlacement1 (int *hex[]*, int * *place*, int *j*, note *noteAr[]*, int * *n*)

```
00244                                                                        {
00245   int i = 3;
00246   while(i < 7 && hex[(j + i++)] > 0x80);
00247   if(checkNextEvent(hex, (j + i))){
00248     *place = j;
00249     if(hex[j] == 0x90){
00250       fillNote(hex[j + 1], &noteAr[*n]);
00251       *n += 1;
00252     }
00253   }
00254 }
```

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

```
00256                                                  {
00257   int i = 2;
00258   while(i < 6 && hex[(j + i++)] > 0x80);
00259   if(checkNextEvent(hex, (j + i)))
00260     *place = j;
00261 }
```

4.2.4.20 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.

```
00646                            {
00647   int major[] = {0, 2, 4, 5, 7, 9, 11};
00648
00649   for(int i = 0; i < SCALESIZE; i++){
00650     if(toneLeap == major[i])
00651       return 1;
00652   }
00653   return 0;
00654 }
```

4.2.4.21 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.

```
00632                                {
00633   int minor[] = {0, 2, 3, 5, 7, 8, 10};
00634
00635   for(int i = 0; i < SCALESIZE; i++){
00636     if(toneLeap == minor[i])
00637       return 1;
00638   }
00639   return 0;
00640 }
```

4.2.4.22 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.

```
00603                                                          {
00604   int toneLeap, isMinor = 1, isMajor = 1;
00605
00606   for(int i = 0; i < size; i++){
00607     if(otherTones[i] < keytone)
00608       otherTones[i] += 12;
00609       toneLeap = otherTones[i] - keytone;
00610
00611       if(isMinor)
00612         isMinor = isInMinor(toneLeap);
00613       if(isMajor)
00614         isMajor = isInMajor(toneLeap);
00615     }
00616
00617     if(isMinor && isMajor)
00618       return 0;
00619     else if(isMinor)
00620       return -1;
00621     if(isMajor)
00622       return 1;
00623
00624     return 0;
00625 }
```

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

```
00094                                       {
00095   DIR *dir;
00096   FILE *f;
00097   char MIDIfile[25];
00098   /*Variables*/
00099   int numbersInText = 0, notes, i = 0, size = 0, moodOfMelodi = 0;
00100   /* PLACEHOLDER FIX THIS */
00101   int mode = 5, tempo = 5, toneLength = 5, pitch = 5;
00102   FILE* moods = fopen("moods.txt", "r");
00103   if(moods == NULL){
00104     perror("Error: moods missing ");
00105     exit(EXIT_FAILURE);
00106   }
00107   AMOUNT_OF_MOODS = FindMoodAmount(moods);
00108   moodWeighting moodArray[AMOUNT_OF_MOODS];
00109   data data = {0, major, D};
```

```
00110   if (argv[1] == NULL){
00111     checkDirectory(MIDIfile);
00112     f = fopen(MIDIfile,"r");
00113     if(f == NULL){
00114       perror("Error opening file");
00115       exit(EXIT_FAILURE);
00116     }
00117     closedir (dir);
00118   }
00119   else if(argv[1] != NULL){
00120     f = fopen(argv[1],"r");
00121     if(f == NULL){
00122       perror("Error opening file");
00123       exit(EXIT_FAILURE);
00124     }
00125   }
00126
00127   int *hex = (int *) malloc(CHARS * sizeof(int));
00128   if(hex == NULL){
00129     printf("Memory allocation failed, bye!");
00130     exit(EXIT_FAILURE);
00131   }
00132   /*Reading the data from the file*/
00133   numbersInText = getHex(f, hex);
00134   fillSongData(&data, hex, numbersInText);
00135   notes = countNotes(hex, numbersInText);
00136   note *noteAr = (note*) malloc(notes * sizeof(note));
00137   if(noteAr == NULL){
00138     printf("Memory allocation failed, bye!");
00139     exit(EXIT_FAILURE);
00140   }
00141   eventPlacement placement[numbersInText];
00142   int ticks[numbersInText];
00143   findEvents(numbersInText, hex, placement, noteAr, ticks, &size);
00144   insertMoods(moodArray, moods);
00145   for(i = 0; i < notes; i++)
00146     printNote(noteAr[i]);
00147   findMode(noteAr, notes, &data);
00148   settingPoints(&mode, &tempo, &toneLength, &pitch, data, notes, noteAr, &size);
00149   printSongData(data);
00150   moodOfMelodi = weightingMatrix(moodArray, mode, tempo, toneLength, pitch);
00151   printf("%d\n", moodOfMelodi);
00152
00153
00154   /*Clean up and close*/
00155   fclose(f);
00156   free(hex);
00157   free(noteAr);
00158
00159   return 0;
00160 }
```

4.2.4.24 void printNote (note *note*)

A function to print the note

Parameters

note	the note structure to be printed

```
00339                           {
00340   printf("Tone: ");
00341
00342   switch (note.tone){
00343     case C     : printf("C") ; break;
00344     case Csharp: printf("C#"); break;
00345     case D     : printf("D") ; break;
00346     case Dsharp: printf("D#"); break;
00347     case E     : printf("E") ; break;
00348     case F     : printf("F") ; break;
00349     case Fsharp: printf("F#"); break;
00350     case G     : printf("G") ; break;
00351     case Gsharp: printf("G#"); break;
00352     case A     : printf("A") ; break;
00353     case Asharp: printf("A#"); break;
00354     case B     : printf("B") ; break;
00355     default    : printf("Undefined note"); break;
00356   }
00357   printf(", octave: %d\n", note.octave);
00358 }
```

4.2.4.25 void printSongData (data *data*)

A function to print out the overall data of the song, tempo and mode

Parameters

data	the data to be printed

```
00363                                {
00364   printf("Tempo: %d\nMode: ", data.tempo);
00365   switch(data.mode){
00366     case minor: printf("minor"); break;
00367     case major: printf("major"); break;
00368     default: printf("unknown mode"); break;
00369   }
00370   printf("\nKeytone: %d", data.key);
00371   putchar('\n');
00372 }
```

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

```
00374
            {
00375   int deltaTime = 2, combined = 0, averageNote = 0;
00376   switch(data.mode){
00377     case minor: *mode = -5; break;
00378     case major: *mode = 5; break;
00379     default: *mode = 0; break;
00380   }
00381   if(data.tempo < 60)
00382     *tempo = -5;
00383   else if(data.tempo >= 60 && data.tempo < 70)
00384     *tempo = -4;
00385   else if(data.tempo >= 70 && data.tempo < 80)
00386     *tempo = -3;
00387   else if(data.tempo >= 80 && data.tempo < 90)
00388     *tempo = -2;
00389   else if(data.tempo >= 90 && data.tempo < 100)
00390     *tempo = -1;
00391   else if(data.tempo >= 100 && data.tempo < 120)
00392     *tempo =  0;
00393   else if(data.tempo >= 120 && data.tempo < 130)
00394     *tempo =  1;
00395   else if(data.tempo >= 130 && data.tempo < 140)
00396     *tempo =  2;
00397   else if(data.tempo >= 140 && data.tempo < 150)
00398     *tempo =  3;
00399   else if(data.tempo >= 150 && data.tempo < 160)
00400     *tempo =  4;
00401   else if(data.tempo >=  160)
00402     *tempo =  5;
00403
00404   switch(deltaTime){
00405     case 1: *length = -5; break;
00406     case 2: *length = -4; break;
00407     case 4: *length = -2; break;
00408     case 8: *length =  0; break;
00409     case 16: *length = 3; break;
00410     case 32: *length = 5; break;
00411   }
00412   for (int i = 0; i < notes; i++){
00413     combined += noteAr[i].average;
00414   }
00415   averageNote = combined/notes;
00416
00417   if(averageNote <= 16)
00418     *octave = -5;
00419   else if(averageNote >= 17 && averageNote <= 23)
00420     *octave = -4;
00421   else if(averageNote >= 24 && averageNote <= 30)
00422     *octave = -3;
00423   else if(averageNote >= 31 && averageNote <= 37)
00424     *octave = -2;
00425   else if(averageNote >= 38 && averageNote <= 44)
00426     *octave = -1;
00427   else if(averageNote >= 45 && averageNote <= 51)
00428     *octave = 0;
00429   else if(averageNote >= 52 && averageNote <= 58)
00430     *octave = 1;
00431   else if(averageNote >= 59 && averageNote <= 65)
00432     *octave = 2;
00433   else if(averageNote >= 66 && averageNote <= 72)
00434     *octave = 3;
00435   else if(averageNote >= 73 && averageNote <= 79)
00436     *octave = 4;
00437   else if(averageNote >=80)
```

```
00438     *octave = 5;
00439 }
```

4.2.4.27 int sortResult (const void * *pa*, const void * *pb*)

```
00475                                                {
00476   int a = *(const int*)pa;
00477   int b = *(const int*)pb;
00478   return (b-a);
00479 }
```

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

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

A function to sort integers in ascending order.

```
00507                                               {
00508   int *i1 = (int*) a, *i2 = (int*) b;
00509
00510   return *i1 - *i2;
00511 }
```

4.2.4.30 int weightingMatrix (moodWeighting *moodArray[]*, int *mode*, int *tempo*, int *toneLength*, int *pitch*)

```
00452                                                                                                                {
00453   int result[AMOUNT_OF_MOODS];
00454
00455   for(int i = 0; i < AMOUNT_OF_MOODS; i++){
00456     result[i] = 0;
00457   }
00458
00459   for(int i = 0; i < AMOUNT_OF_MOODS; i++){
00460     result[i] += (moodArray[i].mode * mode);
00461     result[i] += (moodArray[i].tempo * tempo);
00462     result[i] += (moodArray[i].toneLength * toneLength);
00463     result[i] += (moodArray[i].pitch * pitch);
00464   }
00465
00466   for(int i = 0; i < AMOUNT_OF_MOODS; i++){
00467     printf("%s: %d\n", moodArray[i].name, result[i]);
00468   }
00469
00470   qsort(result, AMOUNT_OF_MOODS, sizeof(int), sortResult);
00471   return result[0];
00472 }
```

4.2.5 Variable Documentation

4.2.5.1 int AMOUNT_OF_MOODS

4.3 test.c File Reference

```
#include <stdlib.h>
#include <stdio.h>
```

Functions

- int **main** (void)
- void **testFunk** (void)

4.3.1 Function Documentation

4.3.1.1 int main (void)

```
00003               {
00004   printf("Jonas er en kagemand!\nOg han har lange løg.\n");
00005
00006   return 0;
00007 }
```

4.3.1.2 void testFunk (void)

```
00012                   {
00013   int stuff = 1337;
00014 }
```

Index