BeatSync

מגיש: וולדיסלב פוברז'ני

כיתה: י"ג הנדסת תוכנה

בית ספר: מכללת הכפר הירוק

מנחה: יהודה אור

שפות תיכנות: C#, XML, Arduino

סביבות פיתוח: Visual Studio IDE (Xamarin), Arduino IDE

מבוא

פרויקט זה נועד להעשיר את החויה של ספורטאים והעוסקים בפעילות גופנית סדירה.

העשרת החויה נעשית על יד התאמת קצב המוזיקה, אותה שומע הספורטאי במכשיר הטלפון הנייד שלו לקצב פעימות הלב המדדות בו זמנית. כמו כן מוצגות על מסך הטלפון קצב פעימות הלב והמוסיקה וגם שם השיר הנבחר אוטומטית מתוך רשימה נבחרת.

מטרת הפרויקט היא להראות יכולת שליטה בפיתוח תוכנה במרחב רחב דיסציפלינות הכולל חומרה ותוכנה כמפורט:

- יסודות עיבוד אותות דיגיטליים (DSP – Digital Signal Processing)
- יישום של פונקציות מתמטיקה לעיבוד אותות דיגיתליים – כדוגמא FFT
- עבודה עם המבנה של הפורמט WAVE (כתיבת מפענח) לצורך העיבוד של אודיו
- עבודה בתחום בקרי תכנות Arduino
- בניית מעגלים חשמליים לחיבור כל מרכיבי החומרה של הפרויקט
- יישום פרוטוקול תקשורת Bluetooth לצורך העברת מידע
- יצירת יישום עבור הטלפון החכם על פלטפורמת אנדרואיד הכולל גרפיקה
- מימוש ידע זה באמצעות תכנות (קוד)
- תכנות בשפות C#, XML, Arduino

תוכן עניינים

רפלקציה

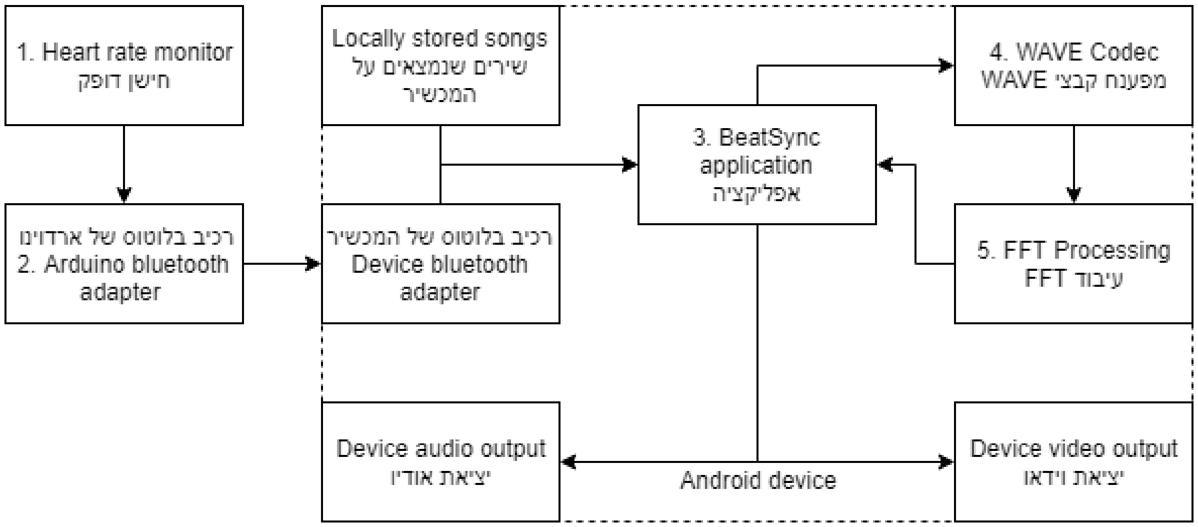
נהנתי מאוד לעבוד על הפרויקט שלי. התחום של מוזיקה והתצוגה שלה באופן דיגיטלי תמיד עניינו וסקרנו אותי. פרויקט זה נתן לי אפשרות לחקור איך התהליך של דיגיטיזציה של שירים שאנו שומעים כל יום עובדת מאחורי הקלעים. יתר על כך, חקירה בתחומים האלו לימדה אותי לעבד את אינפורמציה הדיגיטלית ולחשב מידע נוסף שימושי.

בחרתי את הנושא המסויים הזה לא רק בגלל הסקרנות האישית שלי, אלא גם בגלל שהרעיון של הפרוייקט בולט מפרויקטים רגילים. קיבלתי אפשרות לייצר משהו שלא רק עוזר לי ללמוד ולהתקדם, אבל גם מניע אחרים להצליח במטרות הספורטיביות שלהם.

בזמן של כתיבה וחקירה על הפרויקט יצא לי ללמוד על מימנויות שחדשות לי:

- חקרתי על פורמטים RIFF ו- WAVE, ועל המבנה הבינארי שלהם, איך להוציא מידע שימושי מקבצים האלו לשנות וגם לכתוב מידע זו מחדש לעיבוד על ידי תוכנות אודיו אחרות.
- למדתי לעבד אותות דיגיטליים של אודיו ולהשתמש בנוסחאות מתמטיקה מתקדמות כדי להעביר אותות מדומיין של זמן לדומיין של תדירויות ובחזרה.
- חקרתי בתחום של בקרי ארדוינו, חישנים ומעגלים חשמליים.
- למדתי על עבודה עם אפליקציות על פלטפורמת אנדרואיד ובניה של אפליקציות משלי.
- חקרתי על שידור אלחוטי של מידע דרך בלוטוס, עבודה עם סוקטים והחלפת אינפורמציה בין שרתים ולקוחות.
- פיתחתי אוטודידקטיות – יכולת למידה עצמית ממקורות מידע שונים (כמו אינטרנט).

תאור טכני



תרשים 1

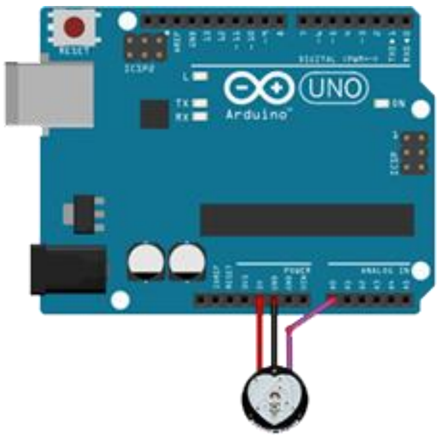
מבחינה טכנית, פרויקט זה הוא מכשיר קטן המכיל חישן דופק מחובר לשלט ארדוינו שמתקשר דרך רכיב בלוטוס עם אפליקציה בטלפון של המשתמש (**תרשים 1**).

מכשיר זה מונח על היד ומודד את הדופק של אדם במהלך פעילות גופנית. תוצאות המדידות מוגשות ליישום בטלפון ומפורשות על המסך. אפליקציה BeatSync סורקת את כל הקבצי מוזיקה בפורמט WAV ומחשבת את הקצב של כל שיר הנמצא על הטלפון של המשתמש.

בהתאם לשינויים בקצב הלב, אפליקציה בוחרת שיר לנגינה עם קצב הכי מתאים לקצב פעימות הלב החדש והמוזיקה משתנה לפי הפלייליסט האישי של האדם המוגדר באפליקציה. לכן, המשתמש יכול לפקח על הדופק שלו, ולשמור אותו בתוך מגבלת הגיל (המוגדות גם בתוכנה) במהלך פעילות גופנית או ספורט, ובו זמנית ליהנות מהמוזיקה האהובה עליו.

אופן הביצוע ורקע תיוארטי

1. חישן דופק



תרשים 2

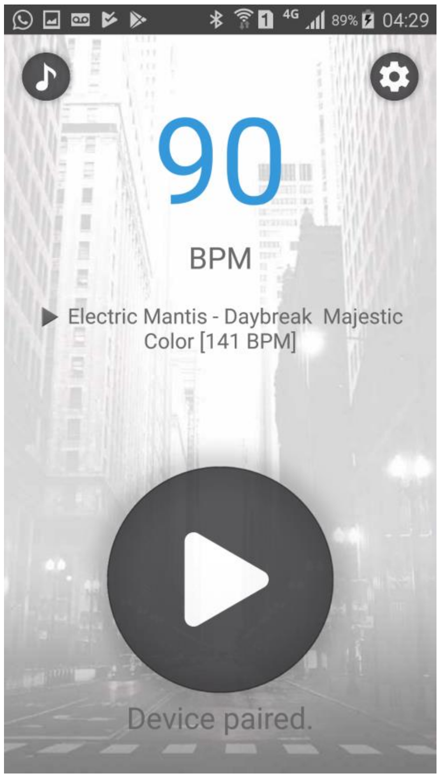
חיישן דופק שנשתמש בו בפרוייקט הזה הוא SEN-11574 מחובר לבקר ארדוינו (**תרשים 2**).

2. רכיב Bluetooth

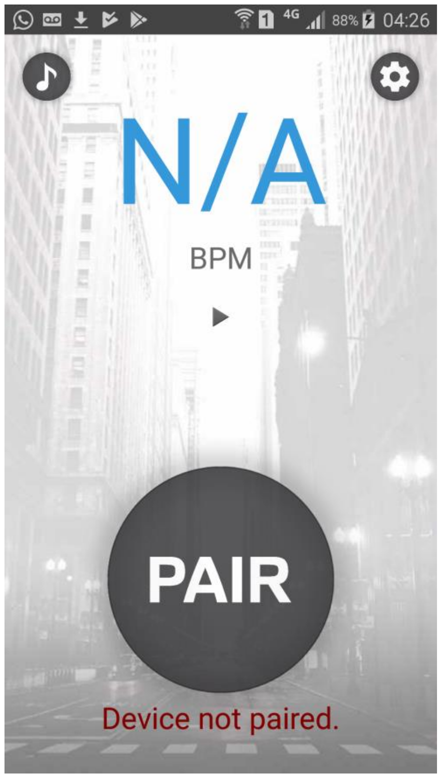
תרשים 3

רכיב בלוטוס המחובר למערכת שלנו הוא HC-05 (**תרשים 3**). חוץ מפינים GND (GROUND) ו-VCC (VOLT) אנו גם נחבר את TXD (TRANSMITTER) שיתן לרכיב אפשרות לשדר מידע.

3. אפליקציה BeatSync



תרשים 4



תרשים 5

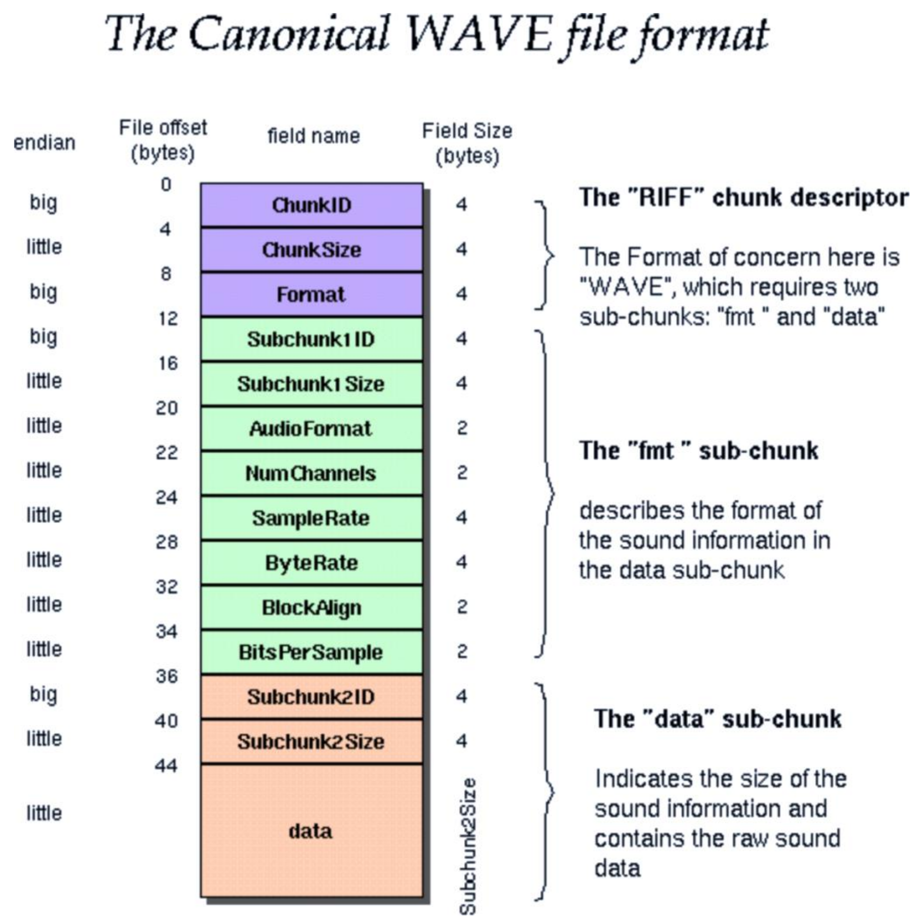
אפליקציה **BeatSync** הוא כלי שמחבר את כל החלקי הפרוייקט ביחד. אפליקציה זו כתובה למכשירי אנדרואיד עם רכיבי Bluetooth המסוגלים לתקשר עם רכיב HC-05 ולקבל מידע מרכיב זה. תוכנה זו גם אחראית על סריקת השירים בטלפון של המשתמש בפורמט WAVE, חישוב הקצב של כל שיר והתאמה של שיר המתנגן לקצב פעימות הלב של המשתמש בזמן פעילות גופנית.

לאפליקציה יש שני מצבים –Paired (**תרשים 4**) ו-Not Paired (**תרשים 5**). בשני המצבים המשתמש יכול לקבל מידע על הקצב של כל שיר בפורמט WAVE הנמצא על המכשיר שלו ולנגן אותם. אחרי הלחיצה על כפתור הנגינה במצב Paired (כשקיים קשר בין חישן הדופק והמכשיר של המשתמש) אפליקציה תשלוט בנגינה של השירים לפי מידע המתקבל מרכיב הבלוטוס של הארדוינו.

אפליקציה תפסיק את נגינת המוסיקה ותתריע על המעבר העליון של קצב פעימות הלב של המשתמש, טווח המוגדר ע"פ גיל המשתמש. אחרי החזרת הקצב לטווח המותר האפליקציה תחזור לנגן מוסיקה.

כשאפליקציה לא נמצאת במצב של של נגינה, למשתמש יש שליטה רגילה והוא יכול נגן כל שיר נבחר – המשתמש יכול לבחור איזה שירים מותר לאפליקציה לנגן בזמן פעולות פיזיות (לבחור פלייליסט), לנגן שירים שעברו עיבוד על ידי המפענח, לעצור ולהמשיך נגינה של השיר בכל נקודת זמן.

4. פורמט WAVE

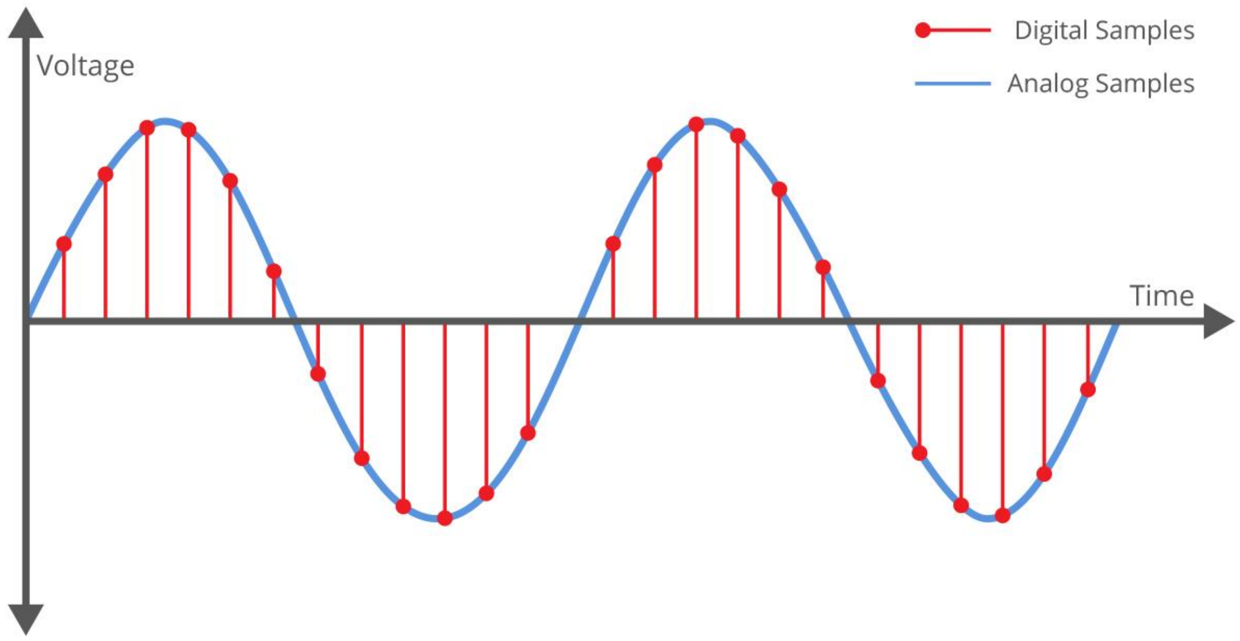


תרשים 6

פורמט WAVE הוא פורמט אודיו רשמי של מיקרוסופט מפותח באמצעות שיטה של שמירת מידע **RIFF** (**R**esource **I**nterchange **F**ile **F**ormat). שיטה זו מחלקת מידע לכמה "נתחים" (chunks). בניגוד לפורמט AVI המפותח באמצעות RIFF המכיל בתוכו גם וידאו ואודיו, פורמט WAVE יכול להחזיק רק מידע קולית.

קבצים WAVE חייבים להכיל שלושת הנתחים הנתונים ב**תרשים 6**: "RIFF", "fmt ", ו- "data". נתח "RIFF" מגדיר את הקובץ כתור קובץ RIFF, נותן מידע על גודל הקובץ ומגדיר סוג קובץ מסוג WAVE. נתח "fmt " מכיל מידע על נתונים גולמיים השמורים בקובץ. מידע זה מכיל אינפורמציה חשובה עבור השמעת אודיו כמו קצב דגימה (Sample Rate), קצב נתונים (Bit Rate) וכו'. נתח "data" שומר את כל הנתונים הגולמיים – מידע בכי חשוב בקובץ שנצטרך לעבד בהמשך.

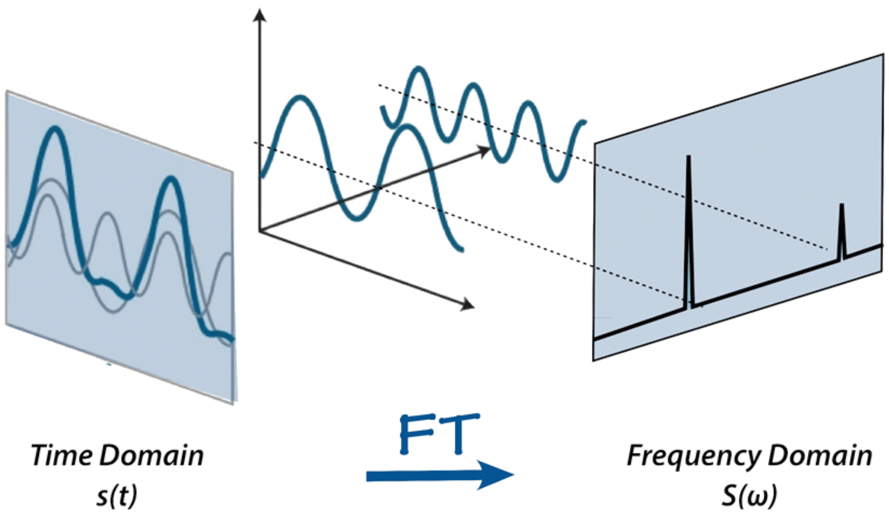
דגימה ו-DSP – עיבוד אותות דיגיטליים



תרשים 7

DSP (**D**igital **S**ignal **P**rocessing) הוא תחום העוסק באותות דיגיטליים ועיבוד של אותות אלה. אחרי שאות אנלוגי עובר תהליך של דיגיטליזציה בשם **דגימה** (Sampling), הוא מתחלק לדגימות שמחקות את הצורה של אות הנקלט (**תרשים 7**) ונהיה מוכן לעיבוד DSP. ניתן לעבד אותות אודיו דיגיטליים באמצעות כלים שונים – מיקסרים (מערבלים), אקולייזרים (משווים), פילטרים (מסננים) וכו'. לצורך החישוב הקצב של שירים נצטרך לבנות מסנן משלנו.

5. DFT vs FFT – התמרת פורייה



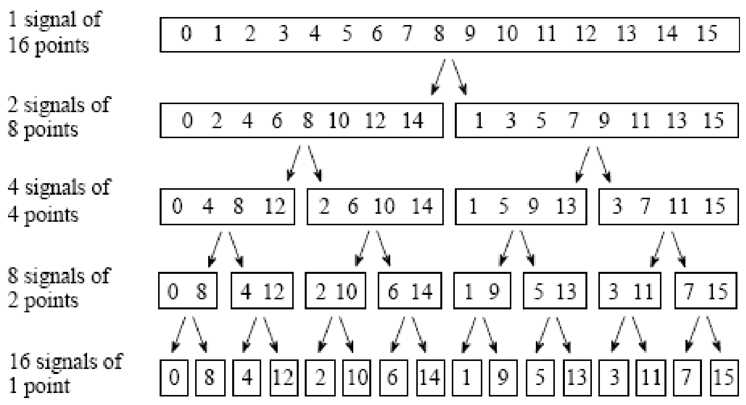
תרשים 8

FT (**F**ourier **T**ransform) היא התמרה שמפרקת פונקציות של זמן (אותות) לתדרים שמרכיבים אותן. התמרת פורייה היא אחד הכלים החשובים בעיבוד אותות אודיו שמאפשר להעביר אות מתחום הזמן לתחום התדר (**תרשים 8**) ונותן לנו שליטה בתדירויות שמרכיבים את האות הנתון.

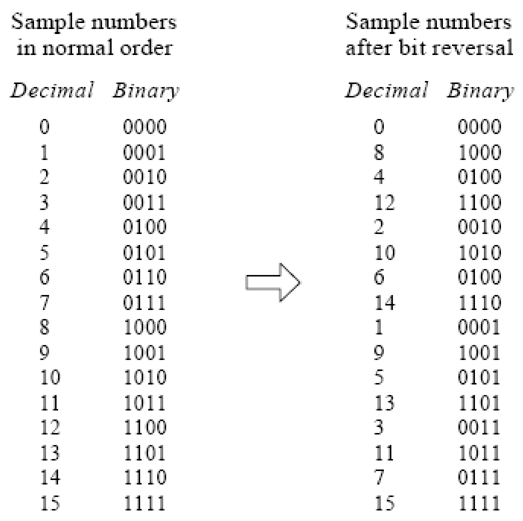
$$X(k)=\sum_{n=0}^{N-1}x(n)e^{-i2\pi kn/N}$$

תרשים 9

התמרת פורייה מתמשכת נקראת **DFT** (**D**iscrete **F**ourier **T**ransform) המאפשרת לעבד אות באורך N דגימות. את ההתמרה DFT ניתן לממש עם פונקציה מתמטית נתונה ב**תרשים 9**. חסרון של וריאציה הזו של התמרת פורייה הוא שסיבוכיות זמן ריצה של אלגוריתם DFT ממומש מקוד היא O(n^2). משך זמן החישוב תלוי בכמות הדגימות וגודל בצורה אקספוננציאלית – חישוב DFT עבור שיר של 5 דקות יכול לקחת שעות.

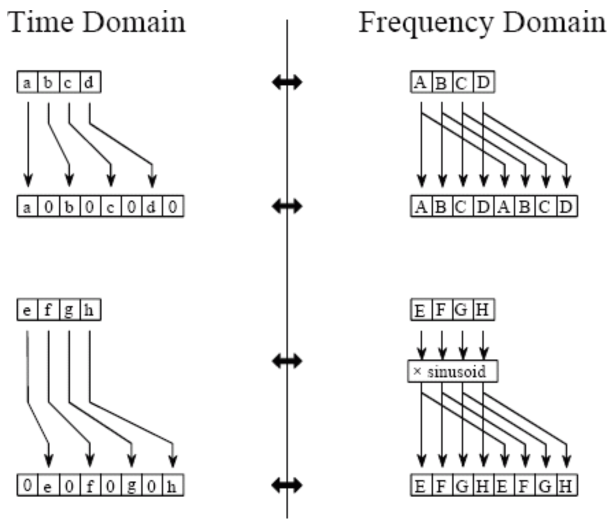


תרשים 10



Sample numbers in normal order			Sample numbers after bit reversal	
Decimal	*Binary*		*Decimal*	*Binary*
0	0000		0	0000
1	0001		8	1000
2	0010		4	0100
3	0011		12	1100
4	0100		2	0010
5	0101		10	1010
6	0110	⇨	6	0100
7	0111		14	1110
8	1000		1	0001
9	1001		9	1001
10	1010		5	0101
11	1011		13	1101
12	1100		3	0011
13	1101		11	1011
14	1110		7	0111
15	1111		15	1111

תרשים 11



תרשים 12

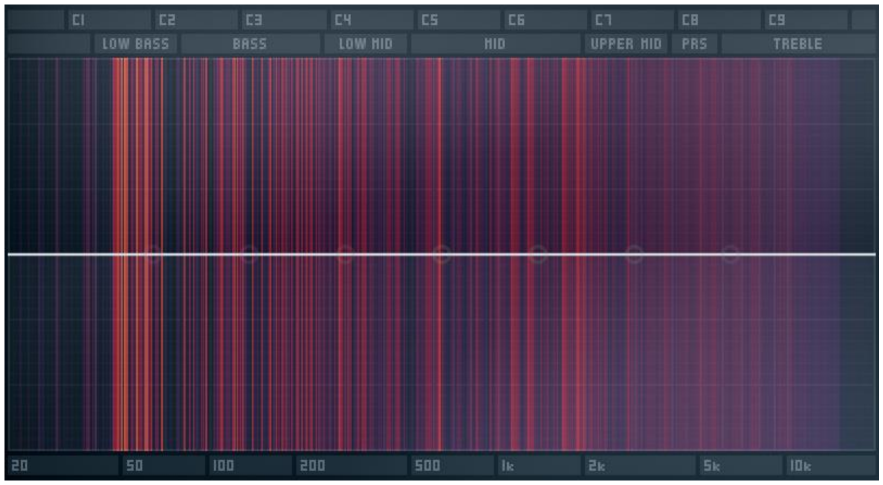
כדי לקטן את זמני החישוב שלנו, נשתמש בוריאציה אחרת של התמרת פורייה בשם **FFT** (**F**ast **F**ourier **T**ransform). אלגוריתם FFT עובד באופן הבא:

1) חלוקה של כל האות לאותות קטנים יותר (**תרשים 10**)
2) מיון של חלקים החדשים עם אלגוריתם בשם Bit Reversal (**תרשים 11**)
3) חישוב DFT עבור החלקים האלו בנפרד
4) שילוב של ספקטרום באופן בדיוק הפוך מאופן שבו פרקנו את האות בתחום הזמן (**תרשים 12**)

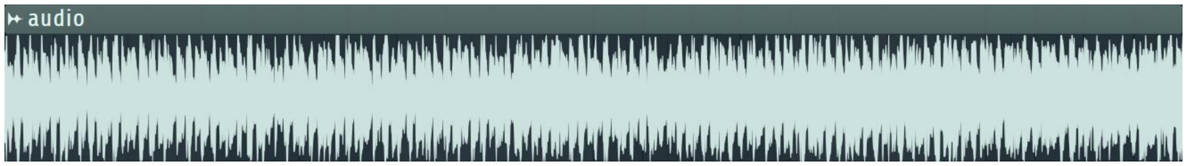
סיבוכיות זמן ריצה של אלגוריתם FFT היא O(n * log (n)). נשים את זה בפרספקטיבה – עבור חישוב של 1024 דגימות, אלגוריתם FFT הוא פי 102.4 יותר מהר מ-DFT. שיפור ביעילות אלגוריתם מאפשר לנו לחשב תחום התדירויות עבור שירים שלמים תוך שניות! למרות התוצאות משופרות שקיבלנו, התמרה FFT היא לא מושלמת – שימוש באלגוריתם זה מגביל את כמות הדגימות הזמינות לחישוב

לחזקה של 2. זאת אומרת שאנו מסוגלים לעבד אותות אך ורק באורך של 2^n דגימות. לדוגמה, רק אותות באורך של 256 (2^8), 1024 (2^10), 65536 (2^16) וכו' דגימות נתונים לחישוב על ידי התמרה FFT. קיימות שיטות שונות שעוקפות את ההגבלה זו (כמו ריפוד אות עם אפסים עד החזקה הקרובה ביותר של 2), אך כל שיטה פוגעת ביעילות וזמן עיבוד של אלגוריטם. לכן, כדי לחשב תחום התדר מהר יותר, נצטרך לעבד אותות קטנים יותר.

כתיבת מסנן אודיו



תרשים 13



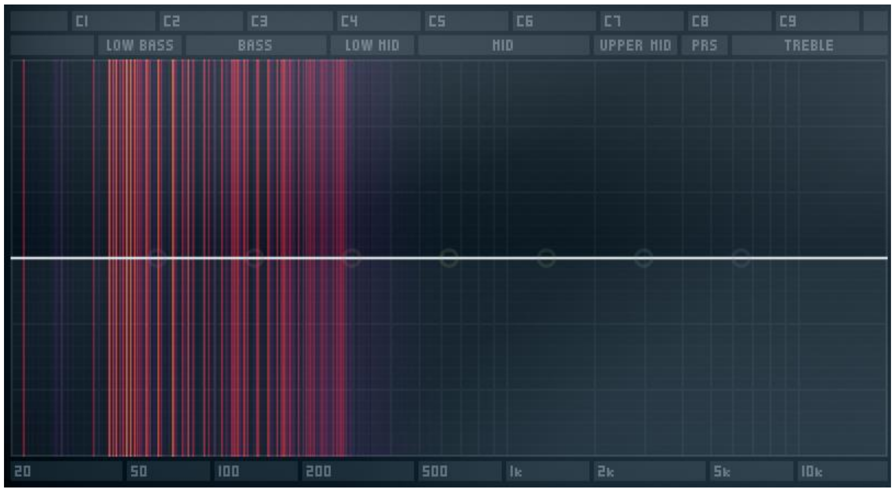
תרשים 14

מסנן אודיו (Audio Filter) הוא כלי עיבוד אותות דיגיטליים בסיסי. תפקידו הראשי של מסנן אודיו הוא לקבל אות ולסנן טווח תדירויות מסוים המוגדר בהגדרות של המסנן, אף כי מסננים גם מסוגלים להגביר ולהחליש תדירויות המתקבלות מהאות. רוב המסנני אודיו מיקצועים עובדים עם טווח של תדירויות מוגבל בין 0 – 22000 הרץ בגלל שהטווח תדרים של שמיעה אנושית (טווח שהאוזניים שלנו מסוגלים לקלוט) הוא בין 20 – 20000 הרץ.

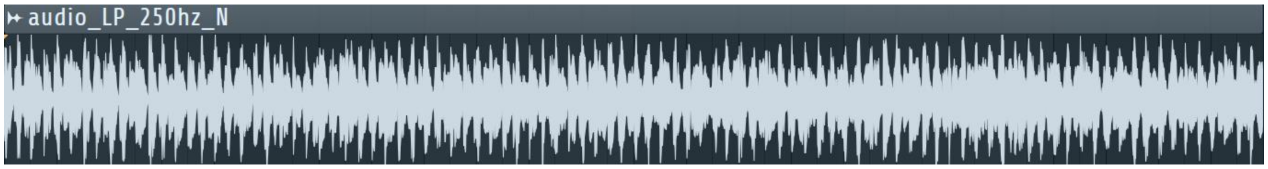
בגלל שמסנני אודיו עובדים עם תחום התדרים, הם חייבים לעבד אותות ולהעביר אותם מתחום הזמן. קיימות פונקציות שמסוגלות לבצע תפקידים בסיסיים של המסנן בתחום הזמן, אך רוב המסננים משתמשים בהתמרת פורייה מכיון שאלגוריטם זה נותן למשתמשים מגוון רחב יותר של פונקציונליות.

ב**תרשים 13** אפשר לראות את הטווח תדירויות של שיר מתנגן. תרשים זה הוא גרף של תדירות כפונקציה של עוצמה – תחום התדר של השיר. ב**תרשים 14** אפשר לראות את צורת הגל של השיר. תרשים זה מציג את הזמן כפונקציה של עוצמה – תחום הזמן של השיר.

תופים הם אלמנט מוסיקלי בשיר שמגדיר את קצב של השיר. טווח תדרים של תופים הוא בין 20 – 210 הרץ. אם אנו לבודד את התדרים האלה עם מסנן אודיו, אנו נבטל את האלמנטים אחרים חסרי תועלת עבורנו ונוכל להתרכז רק בתופים.



תרשים 15



תרשים 16

תרשים 15 מראה לנו את התחום התדרים של השיר אחרי התהליך של סינון תדירויות מאל 250 הרץ.
תרשים 16 מראה לנו את צורת הגל של השיר אחרי סינון. כל פסגה שאנו רואים בתרשים זה הוא תוף המתנגן בשיר. אם נמצא בגרף הזה תוף שחוזר על עצמו לאחר אותה כמות זמן, אפשר לחשב לפי ההבדלים בזמן של התוף את הקצב שלו בפורמט BPM ולפי נתון הזה לחשב את הקצב של השיר עצמו.

נספחים

1. קוד הפרוייקט – Arduino IDE

```
// Set-up low-level interrupts for most accurate BPM math.
#define USE_ARDUINO_INTERRUPTS true
#include <PulseSensorPlayground.h>

// Define variables
PulseSensorPlayground pulseSensor;
int nPulsePin = A0;
int nLEDPin = 13;
int nThreshold = 675;

//-----------------------------------------------------------------------------
//                                  BeatSync
//                                  --------
//
// General  : This code is a part of the BeatSync project and is written
//            for Arduino Nano with a bluetooth adapter and a hearbeat sensor.
//
// Input    : User lays their finger on the hearbeat sensor. Sensor then
//            calculates the user's BPM based on the difference between the
//            impulses that go above the given threshold.
//
// Output   : Program takes the calculated BPM and transmits it as a byte
//            over bluetooth.
//
//-----------------------------------------------------------------------------
// Programmer : Vlad Poberezhny
// Date : 03.04.2018
//-----------------------------------------------------------------------------
void setup()
{
  // Begin serial at 9600 baud
  // Bluetooth adapter can only transmit at 9600 baud or lower
  Serial.begin(9600);

  // Set-up the pulse sensor
  pulseSensor.analogInput(nPulsePin);
  pulseSensor.blinkOnPulse(nLEDPin);
  pulseSensor.setThreshold(nThreshold);

  // Begin reading
  pulseSensor.begin();
}

void loop()
{
  // If heartbeat was detected
  if (pulseSensor.sawStartOfBeat())
  {
    // Convert to BPM, transmit calculation as byte
    Serial.write(pulseSensor.getBeatsPerMinute());
  }

  // Small delay is a good practice
  delay(10);
}
```

MainActivity.cs – BeatSync – קוד הפרוייקט

```
using System.Linq;
using System.Collections.Generic;
using System.IO;
using Android.App;
using Android.Widget;
using Android.OS;
using Android.Views;
using Android.Bluetooth;
using Android.Content;
using Android.Graphics;
using Android.Support.V4.App;
using Android.Content.Res;

namespace BeatSync
{
    [Activity(Label                   = "BeatSync",
              MainLauncher            = true,
              ConfigurationChanges    = Android.Content.PM.ConfigChanges.Orientation |
                                        Android.Content.PM.ConfigChanges.ScreenSize)]

    public partial class MainActivity : Activity
    {
        // Current activity views
        private static TextView     txtBPM;
        private static TextView     txtStatus;
        private static ImageView    btnMain;
        private static ImageView    btnPlaylist;
        private static ImageView    btnSettings;
        private static TextView     txtSong;

        // Adapters and receivers
        private static BluetoothAdapter         PhoneAdapter;
        private static ArduinoBroadcastReceiver Receiver;

        // User related variables
        private static int userAge;
        private static int userMinBPM;
        private static int userMaxBPM;

        // Other variables
        private static float btnMain_DefaultAlpha;

        /// <summary>
        /// Changes the text and color of the status label.
        /// </summary>
        /// <param name="status">New status text.</param>
        /// <param name="color">Color of the new text.</param>
        protected static void SetStatus(string status,
                                        Color color)
        {
            // If status label is initialized
            if (txtStatus != null)
            {
                // Change status text and color
                txtStatus.Text = status;
                txtStatus.SetTextColor(color);
            }
        }
```

```csharp

        /// <summary>
        /// Sets the current age of user.
        /// Calculates and returns the safe BPM range for user.
        /// </summary>
        /// <param name="age">User age.</param>
        /// <returns>Safe BPM range.</returns>
        public static int[] SetAge(int age)
        {
            userAge = age;
            userMaxBPM = (int)((BEATSYNC_CONSTANTS.BEATSYNC_MAXBPM -
    userAge) * 0.85);
            userMinBPM = (int)((BEATSYNC_CONSTANTS.BEATSYNC_MAXBPM -
    userAge) * 0.40);
            return (new int[] { userMinBPM, userMaxBPM });
        }

        /// <summary>
        /// Changes text of the song label.
        /// </summary>
        /// <param name="label">Song title.</param>
        public static void SetSongLabel(string label)
        {
            txtSong.Text = "▶ " + label; // ▌▌ ▶ ⏸▌ ▌  ▌  ▌
        }

        /// <summary>
        /// Function returns the key of the given value in the given dictionary.
        /// </summary>
        /// <param name="dictionary">Given dictionary.</param>
        /// <param name="value">Given dictionary value.</param>
        /// <returns>Dictionary key by value.</returns>
        public static string GetKeyByValue(Dictionary<string, string> dictionary,
                                           string value)
        {
            return (dictionary.FirstOrDefault(item => item.Value.Equals(value)).Key);
        }

        /// <summary>
        /// Called when result for a request has been received.
        /// </summary>
        /// <param name="requestCode">Code of the request.</param>
        /// <param name="resultCode">Result of the request.</param>
        /// <param name="data">Data received.</param>
        protected override void OnActivityResult(int requestCode,
                                                 Result resultCode,
                                                 Intent data)
        {
            // Set main button alpha back to default
            btnMain.Alpha = btnMain_DefaultAlpha;

            // If received result code is "ENABLE BLUETOOTH" code AND result is
"OK"
            if (requestCode == BEATSYNC_CONSTANTS.REQUEST_ENABLE_BLUETOOTH &&
                resultCode == Result.Ok)
            {
                // Change status
                SetStatus("Searching...", BEATSYNC_CONSTANTS.Colors.TextGray);

                // Cancel discovery if device is already discovering
                if (PhoneAdapter.IsDiscovering)
```

```
                        {
                            PhoneAdapter.CancelDiscovery();
                        }

                        // Start discovering
                        new BluetoothDiscoverTask().Execute(PhoneAdapter);
                    }

                    // Otherwise no changes, make button clickable
                    else
                    {
                        btnMain.Clickable = true;
                    }
                }

                /// <summary>
                /// Sends a basic notification to user.
                /// </summary>
                /// <param name="context">Current application environment context.</param>
                /// <param name="contentText">Notification text.</param>
                /// <param name="iconID">Notification icon ID.</param>
                protected static void SendNotification(string contentText,
                                                       int iconID)
                {
                    // Set-up notification
                    NotificationCompat.Builder newNotification = new NotificationCompat.Builder(Application.Context);
                    newNotification.SetPriority((int)NotificationPriority.Max);
                    newNotification.SetDefaults((int)NotificationDefaults.All);
                    newNotification.SetContentTitle(BEATSYNC_CONSTANTS.BEATSYNC_APPLICATION_NAME);
                    newNotification.SetContentText(contentText);
                    newNotification.SetSmallIcon(iconID);

                    // Display notification
                    ((NotificationManager)Application.Context.GetSystemService(NotificationService)).Notify(BEATSYNC_CONSTANTS.BEATSYNC_NOTIFICATION_ID,
                                                                                                            newNotification.Build());
                }

                /// <summary>
                /// Receives a BPM value. Returns a path to song with the closest BPM to received value.
                /// </summary>
                /// <param name="BPM"></param>
                /// <returns></returns>
                protected static string PickSong(int BPM)
                {
                    // Define KVP variable for the answer
                    KeyValuePair<string, int> ClosestSong = new KeyValuePair<string, int>(null, 255);

                    // Iterate over each analyzed song
                    foreach (KeyValuePair<string, int> Song in BEATSYNC_GLOBALS.songBPMs)
                    {
                        if (new Java.IO.File(Song.Key).Exists() &&              // If current song exists
                            BEATSYNC_GLOBALS.userPlaylist.Contains(Song.Key) &&     // If current song is in the playlist
                            !Song.Key.Equals(BEATSYNC_GLOBALS.currentlyPlaying) &&  // If it's not the currently playing song
                            System.Math.Abs(Song.Value - BPM) < ClosestSong.Value)  // If BPM difference is smaller than the current answer's
                        {
                            // Set the current song to be the current answer
                            ClosestSong = new KeyValuePair<string, int>(Song.Key,
                                                                        System.Math.Abs(Song.Value - BPM));
                        }
                    }

                    // Return the path to the song with the closest BPM
                    return (ClosestSong.Key);
                }

                /// <summary>
                /// Special kind of receiver which only looks for the specific Arduino bluetooth adapter.
                /// </summary>
                protected class ArduinoBroadcastReceiver : BroadcastReceiver
                {
                    public override void OnReceive(Context context,
                                                   Intent intent)
                    {
                        // If a device was found through discovery
                        if (intent.Action == BluetoothDevice.ActionFound)
                        {
                            // Retrieve device data from the intent
                            BluetoothDevice Device = (BluetoothDevice)intent.GetParcelableExtra(BluetoothDevice.ExtraDevice);

                            // If found device is our Arduino adapter
                            if (Device.Address == BEATSYNC_CONSTANTS.ARDUINO_ADDRESS)
                            {
                                // Stop adapter discovery, start pairing process
                                PhoneAdapter.CancelDiscovery();
                                SetStatus("Pairing...", BEATSYNC_CONSTANTS.Colors.TextGray);

                                // Try pairing
                                try
                                {
                                    // Get RFCOMM socket from the adapter
                                    BluetoothSocket ArduinoSocket = Device.CreateRfcommSocketToServiceRecord(BEATSYNC_CONSTANTS.ARDUINO_UUID);

                                    // If device is not paired, display pairing PIN on pairing dialog
                                    if (!PhoneAdapter.BondedDevices.Contains(Device))
                                    {
                                        SendNotification("Pairing PIN: " + BEATSYNC_CONSTANTS.ARDUINO_PAIRING_PIN, Resource.Drawable.icon_bluetooth);
                                    }

                                    // Connect to device
                                    ArduinoSocket.Connect();

```

```
                                    // If no exceptions up to this point - devices are connected
                                    BEATSYNC_GLOBALS.bPaired = true;

                                    // BEGIN TASK THAT UPDATES TEXT LABEL WHILE CONNECTION IS STABLE
                                    new BPMTask().ExecuteOnExecutor(AsyncTask.ThreadPoolExecutor, ArduinoSocket);
                                }

                                // If pairing failed
                                catch (Java.IO.IOException)
                                {
                                    SetStatus("Device not paired.", BEATSYNC_CONSTANTS.Colors.Red);
                                    BEATSYNC_GLOBALS.bPaired = false;
                                }
                            }

                            // Make main button clickable
                            btnMain.Clickable = true;
                        }
                    }
                }

                /// <summary>
                /// Task that discovers bluetooth adapters around the device.
                /// </summary>
                protected class BluetoothDiscoverTask : AsyncTask<BluetoothAdapter, int, BluetoothAdapter>
                {
                    /// <summary>
                    /// Set main button icon to "Pairing".
                    /// </summary>
                    protected override void OnPreExecute()
                    {
                        btnMain.SetImageResource(Resource.Drawable.icon_pairing01);
                    }

                    protected override BluetoothAdapter RunInBackground(params BluetoothAdapter[] @params)
                    {
                        // If device is currently discovering, cancel discovery
                        if (@params[0].IsDiscovering)
                        {
                            @params[0].CancelDiscovery();
                        }

                        // Begin discovering bluetooth adapters
                        @params[0].StartDiscovery();

                        // Wait until the discovery boolean is true
                        while (!@params[0].IsDiscovering) ;

                        // Make 12 loops 1 second each while adapter is enabled
                        for (int nSecond = 0;
                            nSecond < 12 &&
                            @params[0].IsEnabled &&
                            @params[0].IsDiscovering;
                            nSecond++)
                        {
                            // Publish current second
                            PublishProgress(nSecond);

                            // Sleep for 1 second
                            System.Threading.Thread.Sleep(1000);
                        }

                        // Return the adapter
                        return @params[0];
                    }

                    /// <summary>
                    /// Animates the main button icon.
                    /// </summary>
                    /// <param name="values">Seconds passed since start of discovery.</param>
                    protected override void OnProgressUpdate(params int[] values)
                    {
                        // If amount of seconds is even, set to first frame
                        if (values[0] % 2 == 0)
                        {
                            btnMain.SetImageResource(Resource.Drawable.icon_pairing01);
                        }

                        // Otherwise set to second frame
                        else
                        {
                            btnMain.SetImageResource(Resource.Drawable.icon_pairing02);
                        }
                    }

                    protected override void OnPostExecute(BluetoothAdapter Adapter)
                    {
                        // If adapter was disabled OR adapter is still discovering
                        if (!Adapter.IsEnabled ||
                            Adapter.IsDiscovering)
                        {
                            // Discovery was unsuccessful
                            Adapter.CancelDiscovery();
                            txtBPM.Text = "N/A";
                            SetStatus("Device not paired.", BEATSYNC_CONSTANTS.Colors.Red);
                            btnMain.SetImageResource(Resource.Drawable.txt_pair);
                        }

                        // If devices are paired
                        else if (BEATSYNC_GLOBALS.bPaired)
                        {
                            // Change program status
                            SetStatus("Device paired.", BEATSYNC_CONSTANTS.Colors.TextGray);
                            btnMain.SetImageResource(Resource.Drawable.icon_play);
                        }

                        // Make main button clickable
                        btnMain.Clickable = true;
                    }
                }

                /// <summary>
                /// Task receives BPM data over given stream and manages
                /// song playback based on BPM average.
                /// </summary>
                protected class BPMTask : AsyncTask<BluetoothSocket, int, bool>
                {
                    // BPM related variables
                    private int averageBPM      = 0;
                    private int summatedBPM     = 0;
                    private int updateCounter   = 0;

                    /// <summary>
                    /// Receives the BPM data over the given stream.
                    /// Passes each new BPM value to 'PublishProgress'.
                    /// </summary>
                    /// <param name="params">Data stream.</param>
                    /// <returns></returns>
                    protected override bool RunInBackground(params BluetoothSocket[] @params)
                    {
                        // Set-up the adapter input stream
                        Stream ArduinoStream = @params[0].InputStream;

                        // Try communicating with the adapter
                        try
                        {
                            // If stream is readable
                            if (ArduinoStream.CanRead)
                            {
                                // While stream is readable AND devices are connected
                                while (ArduinoStream.CanRead &&
                                       BEATSYNC_GLOBALS.bPaired)
                                {
                                    // While program is passive, publish BPM values
                                    while (ArduinoStream.CanRead &&
                                           BEATSYNC_GLOBALS.bPaired &&
                                           !BEATSYNC_GLOBALS.bActive)
                                    {
                                        PublishProgress(ArduinoStream.ReadByte());
                                    }

                                    // If program became active
                                    if (ArduinoStream.CanRead &&
                                        BEATSYNC_GLOBALS.bPaired &&
                                        BEATSYNC_GLOBALS.bActive)
                                    {
                                        // Set the first received value as the first average BPM
                                        averageBPM = ArduinoStream.ReadByte();

                                        // Run the following command on main UI thread
                                        ((Activity)txtBPM.Context).RunOnUiThread(() =>
                                        {
                                            // Play song with the closest BPM to the average value
                                            Playlist.SetSong(PickSong(averageBPM));
                                        });

                                        // Nullify summation values
                                        updateCounter = summatedBPM = 0;
                                    }

                                    // While stream is readable AND program is active, keep publishing BPM
                                    while (ArduinoStream.CanRead &&
                                           BEATSYNC_GLOBALS.bPaired &&
                                           BEATSYNC_GLOBALS.bActive)
                                    {
                                        PublishProgress(ArduinoStream.ReadByte());
                                    }
                                }
                            }
                        }

                        // If connection with Arduino was terminated
                        catch (Java.IO.IOException)
                        {
                            return (true);
                        }

                        return (false);
                    }

                    /// <summary>
                    /// Updates the BPM label with the data received over bluetooth.
                    /// If program is active, manages song selection.
                    /// Labels can be updated here since this procedure is on the UI thread.
                    /// </summary>
                    /// <param name="values">New BPM value.</param>
                    protected override void OnProgressUpdate(params int[] values)
                    {
                        // Update BPM label
                        txtBPM.Text = values[0].ToString();

                        // Add value to BPM summation
                        summatedBPM += values[0];

                        // If enough check have been made AND program is active AND new average differs by at least 10 BPM
                        // OR song has reached the outro
                        if ((++updateCounter >= 120 && BEATSYNC_GLOBALS.bActive && System.Math.Abs(averageBPM - (summatedBPM / (float)updateCounter)) > 10) ||
                            BEATSYNC_GLOBALS.currentSong.CurrentPosition > BEATSYNC_GLOBALS.currentSong.Duration - (int)(BEATSYNC_GLOBALS.currentSong.Duration / BEATSYNC_CONSTANTS.BEATSYNC_SONGCUE))
                        {
                            // If songs are currently not being crossfaded
                            if (!BEATSYNC_GLOBALS.bCrossfading)
                            {
                                // Update average BPM value
                                averageBPM = (int)(summatedBPM / (float)updateCounter);

                                // Play song with the closest BPM to current average BPM
                                Playlist.SetSong(PickSong(averageBPM));

                                // Nullify counter and summation
                                updateCounter = summatedBPM = 0;
                            }
                        }
```

```
                }

                /// <summary>
                /// Called on disrupted connection with Arduino adapter.
                /// </summary>
                /// <param name="bException">If task ended with an exception.</param>
                protected override void OnPostExecute(bool bException)
                {
                    // Set program back to passive
                    BEATSYNC_GLOBALS.bActive = BEATSYNC_GLOBALS.bPaired = false;
                    SetStatus("Device not paired.", BEATSYNC_CONSTANTS.Colors.Red);

                    txtBPM.Text = "N/A";
                    btnMain.SetImageResource(Resource.Drawable.txt_pair);
                }
            }

            /// <summary>
            /// Function is called when device is rotated.
            /// </summary>
            /// <param name="newConfig">New configuraion.</param>
            public override void OnConfigurationChanged(Configuration newConfig)
            {
                // Execute default configuration commands
                base.OnConfigurationChanged(newConfig);

                // If device is turned horizontally
                if (newConfig.Orientation == Android.Content.Res.Orientation.Landscape)
                {
                    // Hide unneeded views
                    btnMain.Visibility = ViewStates.Gone;
                    txtStatus.Visibility = ViewStates.Gone;
                }

                // If device is turned vertically
                else
                {
                    // Show hidden views
                    btnMain.Visibility = ViewStates.Visible;
                    txtStatus.Visibility = ViewStates.Visible;
                }
            }

            protected override void OnCreate(Bundle savedInstanceState)
            {
                // Default creation procedure
                base.OnCreate(savedInstanceState);

                // Set view to "Main" activity
                SetContentView(Resource.Layout.Main);

                // If age is stored, load it
                if (new Java.IO.File(FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERDATA).Exists())
                {
                    using (var streamReader = new StreamReader(System.IO.Path.Combine(FilesDir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERDATA)))
                    {
                        SetAge((userAge = (char)streamReader.Read()));
                    }
                }

                // Otherwise set to default
                else
                {
                    SetAge((userAge = BEATSYNC_CONSTANTS.BEATSYNC_DEFAULTAGE));
                    using (var streamWriter = new StreamWriter(System.IO.Path.Combine(FilesDir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERDATA)))
                    {
                        streamWriter.Write((char)userAge);
                    }

                    // Jump to "Settings" screen
                    StartActivity(typeof(Settings));
                }

                // If there are analyzed and stored songs, load them
                if (new Java.IO.File(FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERSONGS).Exists())
                {
                    using (var streamReader = new StreamReader(System.IO.Path.Combine(FilesDir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERSONGS)))
                    {
                        while (!streamReader.EndOfStream)
                        {
                            BEATSYNC_GLOBALS.songBPMs.Add(streamReader.ReadLine(), int.Parse(streamReader.ReadLine()));
                        }
                    }
                }

                // If there is a stored playlist, load it
                if (new Java.IO.File(FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERPLAYLIST).Exists())
                {
                    using (var streamReader = new StreamReader(System.IO.Path.Combine(FilesDir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERPLAYLIST)))
                    {
                        while (!streamReader.EndOfStream)
                        {
                            BEATSYNC_GLOBALS.userPlaylist.Add(streamReader.ReadLine());
                        }
                    }
                }

                // Initialize views
                btnMain     = FindViewById<ImageView>(Resource.Id.btnMain);
                txtBPM      = FindViewById<TextView>(Resource.Id.BPM);
                txtStatus   = FindViewById<TextView>(Resource.Id.txtStatus);
                btnPlaylist = FindViewById<ImageView>(Resource.Id.imgPlaylist);
                txtSong     = FindViewById<TextView>(Resource.Id.txtSong);
                btnSettings = FindViewById<ImageView>(Resource.Id.imgSettings);

                // Set-up adapter and receiver
                PhoneAdapter    = BluetoothAdapter.DefaultAdapter;
                Receiver        = new ArduinoBroadcastReceiver();
                RegisterReceiver(Receiver, new IntentFilter(BluetoothDevice.ActionFound));

                // Set text colors
                FindViewById<TextView>(Resource.Id.txtBPM).SetTextColor(BEATSYNC_CONSTANTS.Colors.TextGray);
                txtSong.SetTextColor(BEATSYNC_CONSTANTS.Colors.TextGray);

                // On song title click
                txtSong.Click += delegate
                {
                    // If program is passive
                    if (!BEATSYNC_GLOBALS.bActive)
                    {
                        // If song is being played, pause it
                        if (BEATSYNC_GLOBALS.currentSong.IsPlaying)
                        {
                            BEATSYNC_GLOBALS.currentSong.Pause();
                        }

                        // Otherwise resume it
                        else
                        {
                            BEATSYNC_GLOBALS.currentSong.Start();
                        }
                    }
                };

                // On "Playlist" button click
                btnPlaylist.Click += delegate
                {
                    // Start "Playlist" activity
                    StartActivity(typeof(Playlist));
                };

                // On "Settings" button click
                btnSettings.Click += delegate
                {
                    // Start "Settings" activity
                    StartActivity(typeof(Settings));
                };

                // If text of the BPM value was changed
                txtBPM.AfterTextChanged += delegate
                {
                    // If devices are paired AND new BPM value is above the limit
                    if (BEATSYNC_GLOBALS.bPaired &&
                        int.Parse(txtBPM.Text) > userMaxBPM)
                    {
                        // Set color to red
                        txtBPM.SetTextColor(BEATSYNC_CONSTANTS.Colors.Red);

                        // If program is active AND music is playing
                        if (BEATSYNC_GLOBALS.bActive &&
                            BEATSYNC_GLOBALS.currentSong.IsPlaying)
                        {
                            // Stop the music and notify user of high heart rate
                            BEATSYNC_GLOBALS.currentSong.Stop();
                            Android.Media.MediaPlayer.Create(this, Android.Net.Uri.Parse("system/media/audio/ui/LowBattery.ogg")).Start();
                        }
                    }

                    // If devices are paired AND new BPM value is below the limit
                    else if (BEATSYNC_GLOBALS.bPaired &&
                             int.Parse(txtBPM.Text) < userMinBPM)
                    {
                        // Set color to gray
                        txtBPM.SetTextColor(BEATSYNC_CONSTANTS.Colors.TextGray);
                    }

                    // Otherwise
                    else
                    {
                        // Set color to blue
                        txtBPM.SetTextColor(BEATSYNC_CONSTANTS.Colors.Blue);

                        // If program is active AND music is not playing
                        if (BEATSYNC_GLOBALS.bActive &&
                            !BEATSYNC_GLOBALS.currentSong.IsPlaying)
                        {
                            // Resume the song
                            BEATSYNC_GLOBALS.currentSong.Start();
                        }
                    }
                };

                // Save default alpha value for the main button
                btnMain_DefaultAlpha = btnMain.Alpha;

                // On main button click
                btnMain.Click += delegate
                {
                    // Lower the alpha value of the view
                    btnMain.Alpha -= 0.125f;

                    // If devices are not paired
                    if (!BEATSYNC_GLOBALS.bPaired)
                    {
                        // Disable main button
                        btnMain.Clickable = false;

                        // If bluetooth is disabled
                        if (!PhoneAdapter.IsEnabled)
                        {
                            // Request user to enable bluetooth
                            Intent enableBluetooth = new Intent(BluetoothAdapter.ActionRequestEnable);
                            StartActivityForResult(enableBluetooth, BEATSYNC_CONSTANTS.REQUEST_ENABLE_BLUETOOTH);
                        }

                        // Otherwise send an already positive request
                        else
                        {
                            OnActivityResult(BEATSYNC_CONSTANTS.REQUEST_ENABLE_BLUETOOTH, Result.Ok, new Intent());
                        }
                    }

                    // If devices are paired
                    else
                    {
                        // If program is active
                        if (BEATSYNC_GLOBALS.bActive)
```

```
                            {
                                // Switch program to passive
                                BEATSYNC_GLOBALS.bActive = false;
                                btnMain.SetImageResource(Resource.Drawable.icon_play);
                                SetStatus("Device paired.", BEATSYNC_CONSTANTS.Colors.TextGray);
                            }

                            // If program is passive
                            else
                            {
                                // If there are enough songs in the playlist
                                if (BEATSYNC_GLOBALS.userPlaylist.Count >= BEATSYNC_CONSTANTS.BEATSYNC_MINPLAYLIST)
                                {
                                    // Switch program to active
                                    BEATSYNC_GLOBALS.bActive = true;
                                    btnMain.SetImageResource(Resource.Drawable.icon_pause);
                                    SetStatus("Active.", BEATSYNC_CONSTANTS.Colors.TextGray);
                                }

                                // Otherwise notify user
                                else
                                {
                                    Toast.MakeText(this, "Not enough songs in the Playlist.", ToastLength.Short).Show();
                                }
                            }

                            // Return to the default alpha value of the main button
                            btnMain.Alpha = btnMain_DefaultAlpha;
                        }
                    };

                    // Change status bar color
                    if (Build.VERSION.SdkInt >= BuildVersionCodes.Lollipop)
                    {
                        Window.ClearFlags(WindowManagerFlags.TranslucentStatus);
                        Window.AddFlags(WindowManagerFlags.DrawsSystemBarBackgrounds);
                        Window.SetStatusBarColor(Color.DarkGray);
                    }

                    // Begin analyzing songs in the background
                    new Playlist.SongAnalyzer().ExecuteOnExecutor(AsyncTask.ThreadPoolExecutor, Android.OS.Environment.ExternalStorageDirectory.ToString() + "/Music",
                                                                                                       Android.OS.Environment.ExternalStorageDirectory.ToString() + "/Download");
                }

                protected override void OnDestroy()
                {
                    // Execute default "OnDestroy" commands
                    base.OnDestroy();

                    // Unregister the arduino receiver
                    UnregisterReceiver(Receiver);
                }
            }
        }
```

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/background"
    android:minWidth="25px"
    android:minHeight="25px">
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="10dp"
        android:layout_marginRight="10dp"
        android:layout_marginTop="10dp">
        <ImageView
            android:background="@drawable/button"
            android:src="@drawable/icon_note"
            android:layout_width="50dp"
            android:layout_height="50dp"
            android:alpha="0.875"
            android:id="@+id/imgPlaylist" />
        <ImageView
            android:background="@drawable/button"
            android:src="@drawable/icon_settings"
            android:layout_width="50dp"
            android:layout_height="50dp"
            android:layout_alignParentRight="true"
            android:alpha="0.875"
            android:id="@+id/imgSettings" />
    </RelativeLayout>
    <TextView
        android:text="N/A"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="-25dp"
        android:gravity="center"
        android:textColor="#3498db"
        android:id="@+id/BPM"
        android:textSize="100sp" />
    <TextView
        android:text="BPM"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/txtBPM"
        android:gravity="center"
        android:layout_marginBottom="10dp"
        android:textSize="25sp" />
    <TextView
        android:text="▶ "
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/txtSong"
        android:textSize="17.5sp"
        android:gravity="center"
        android:layout_margin="10dp" />
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```xml
        android:layout_marginLeft="10dp"
        android:layout_marginRight="10dp"
        android:layout_marginTop="10dp"
        android:layout_marginBottom="30dp">
        <ImageView
            android:background="@drawable/button"
            android:src="@drawable/txt_pair"
            android:layout_width="235dp"
            android:layout_height="235dp"
            android:layout_marginBottom="-20dp"
            android:id="@+id/btnMain"
            android:alpha="0.875"
            android:layout_centerHorizontal="true"
            android:layout_above="@+id/txtStatus" />
        <TextView
            android:text="Device not paired."
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textColor="#800000"
            android:id="@+id/txtStatus"
            android:layout_alignParentBottom="true"
            android:gravity="center"
            android:textSize="25sp" />
    </RelativeLayout>
</LinearLayout>
```

קוד הפרוייקט – BeatSync – AndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="BeatSync.BeatSync" android:versionCode="1" android:versionName="1.0" android:installLocation="auto">
    <uses-sdk android:minSdkVersion="21" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <application android:allowBackup="true" android:label="@string/app_name" android:theme="@android:style/Theme.DeviceDefault.Light.NoActionBar" android:icon="@drawable/icon_app"></application>
</manifest>
```

```csharp
using System.IO;
using Android.App;
using Android.OS;
using Android.Graphics;
using Android.Runtime;
using Android.Views;
using Android.Widget;

namespace BeatSync
{
    [Activity(Label = "Playlist")]

    public class Playlist : Activity
    {
        // Current activity views
        private static ListView     songList;
        private static ProgressBar  bpmProgressBar;
        private static ImageView    btnBack;

        /// <summary>
        /// Tries to save the playlist in its current state to disk.
        /// </summary>
        /// <returns>True if saved successfully, false otherwise.</returns>
        public static bool SavePlaylist()
        {
            // Try storing playlist on disk
            try
            {
                using (var streamWriter = new StreamWriter(System.IO.Path.Combine(Application.Context.FilesDir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERPLAYLIST), false))
                {
                    foreach (string songPath in BEATSYNC_GLOBALS.userPlaylist)
                    {
                        streamWriter.WriteLine(songPath);
                    }
                }

                return (true);
            }

            // If file is unavailable for writing, don't save
            catch (System.IO.IOException)
            {
                return (false);
            }
        }

        /// <summary>
        /// Receives a path to the next song and crossfades the current song with the next song.
        /// </summary>
        protected class Crossfade : AsyncTask<string, bool, bool>
        {
            protected override void OnPreExecute()
            {
                // Set crossfading boolean
                BEATSYNC_GLOBALS.bCrossfading = true;
            }

            protected override bool RunInBackground(params string[] @params)
            {
                // Create a next song player and a crossfade timer
                Android.Media.MediaPlayer nextSong = Android.Media.MediaPlayer.Create(Android.App.Application.Context,
                                                                                      Android.Net.Uri.Parse(@params[0]));
                System.Diagnostics.Stopwatch faderTimer = new System.Diagnostics.Stopwatch();

                // Set up the next song
                nextSong.SeekTo((int)(nextSong.Duration / BEATSYNC_CONSTANTS.BEATSYNC_SONGCUE));
                nextSong.SetVolume(0.0f, 0.0f);

                // Begin playback and crossfade
                faderTimer.Start();
                nextSong.Start();

                // Crossfade until time limit has passed
                while (faderTimer.ElapsedMilliseconds < 5000)
                {
                    float currentVolume = faderTimer.ElapsedMilliseconds / 5000f;
                    nextSong.SetVolume(currentVolume, currentVolume);
                    BEATSYNC_GLOBALS.currentSong.SetVolume(1f - currentVolume, 1f - currentVolume);
                }

                // Stop the previous song
                BEATSYNC_GLOBALS.currentSong.Stop();

                // Next song is now the currently playing song
                BEATSYNC_GLOBALS.currentSong = nextSong;

                return (true);
            }

            protected override void OnPostExecute(bool result)
            {
                // Set crossfading boolean
                BEATSYNC_GLOBALS.bCrossfading = false;
            }
        }

        /// <summary>
        /// Crossfades to the new song and updates the song label.
        /// </summary>
            /// <param name="path">Path to a new song.</param>
            public static void SetSong(string path)
            {
                new Crossfade().Execute(path);
                BEATSYNC_GLOBALS.currentlyPlaying = path;
                MainActivity.SetSongLabel(BEATSYNC_GLOBALS.songTitles[path]);
            }

            /// <summary>
            /// Background task that analyzes every .WAV file in the given paths.
            /// Task calculates the BPM of every "wave" file not previously analyzed.
```

```
                /// Task then saves the calculation to song dictionary.
                /// </summary>
                public class SongAnalyzer : AsyncTask<string, bool, bool>
                {
                    /// <summary>
                    /// Receives the paths to look for .WAV files in.
                    /// Analyzes each "wave" file and saves to list of songs.
                    /// </summary>
                    /// <param name="params">Paths to .WAV files.</param>
                    /// <returns>Returns true.</returns>
                    protected override bool RunInBackground(params string[] @params)
                    {
                        // Iterate over each given directory
                        for (int nIndex = 0;
                            nIndex < @params.Length;
                            nIndex++)
                        {
                            // For each object in the directory
                            foreach (Java.IO.File File in new Java.IO.File(@params[nIndex]).ListFiles())
                            {
                                // If object is a .WAV file AND its duration will suffice for BPM calculation
                                if (File.IsFile &&
                                    WAV.IsWAV(File.Path) &&
                                    Android.Media.MediaPlayer.Create(Application.Context, Android.Net.Uri.Parse(File.Path)).Duration >= WAV.BPM_CALCULATION_DURATION_MINIMUM * 1000)
                                {
                                    // If BPM for this song was NOT previously calculated
                                    if (!BEATSYNC_GLOBALS.songBPMs.ContainsKey(File.Path))
                                    {
                                        // Load current song to the codec
                                        WAV waveFile = new WAV(File.Path);

                                        // Calculate BPM and add to song dictionary
                                        BEATSYNC_GLOBALS.songBPMs.Add(waveFile.Path, waveFile.BPM);
                                        BEATSYNC_GLOBALS.userPlaylist.Add(waveFile.Path);

                                        // Write calculations to disk
                                        using (var songWriter = new StreamWriter(System.IO.Path.Combine(Application.Context.FilesDir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERSONGS), true))
                                        {
                                            songWriter.WriteLine(waveFile.Path);
                                            songWriter.WriteLine(waveFile.BPM.ToString());
                                        }

                                        // Add new song to playlist
                                        using (var playlistWriter = new StreamWriter(System.IO.Path.Combine(Application.Context.FilesDir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERPLAYLIST), true))
                                        {
                                            playlistWriter.WriteLine(waveFile.Path);
                                        }
                                    }

                                    // Set title for the current song
                                    BEATSYNC_GLOBALS.songTitles.Add(File.Path, File.Name.Substring(0, File.Name.Length - 4) + " [" + BEATSYNC_GLOBALS.songBPMs[File.Path] + " BPM]");

                                    // Add current song to list of available songs
                                    PublishProgress(true);
                                    System.Threading.Thread.Sleep(50);
                                }
                            }
                        }

                        return (true);
                    }

                    /// <summary>
                    /// Receives a single ArrayAdapter with currently available songs.
                    /// Sets the received adapter as the main song list adapter.
                    /// </summary>
                    /// <param name="values">Adapter with available songs.</param>
                    protected override void OnProgressUpdate(params bool[] values)
                    {
                        // Refresh list of songs on new song addition
                        RefreshSongList();
                    }

                    protected override void OnPostExecute(bool result)
                    {
                        // If progress bar is initialized, hide it
                        if (bpmProgressBar != null)
                        {
                            bpmProgressBar.Visibility = ViewStates.Gone;
                        }

                        // Save user playlist
                        SavePlaylist();
                        BEATSYNC_GLOBALS.bSongsAnalyzed = true;
                    }
                }

                /// <summary>
                /// Refreshes the list of analyzed and stored songs.
                /// </summary>
                private static void RefreshSongList()
                {
                    // If stored list of calculated songs exists AND song list view is initialized
                    if (new Java.IO.File(Application.Context.FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERSONGS).Exists() &&
                        songList != null)
                    {
                        // Create a new list and add the song paths to the list
                        JavaList<string> analyzedSongs = new JavaList<string>();
                        using (var songReader = new StreamReader(System.IO.Path.Combine(Application.Context.FilesDir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERSONGS), true))
                        {
                            // Read until the end of file
                            while (!songReader.EndOfStream)
                            {
```

```
                // Save current song path
                string currentPath = songReader.ReadLine();

                // If path is located in the dictionary of song titles, add to analyzed songs
                if (BEATSYNC_GLOBALS.songTitles.ContainsKey(currentPath))
                {
                  analyzedSongs.Add(BEATSYNC_GLOBALS.songTitles[currentPath]);
                }

                // Skip the BPM value for this song
                songReader.ReadLine();
              }
            }

            // Set-up custom adapter with calculated songs to song listview
            songList.Adapter = new ArrayAdapter(songList.Context,
                                                Android.Resource.Layout.SimpleListItemMultipleChoice,
                                                analyzedSongs);

            // Check song if it's in the playlist
            for (int nSongIndex = 0;
            nSongIndex < songList.Count;
            nSongIndex++)
            {
              songList.SetItemChecked(nSongIndex, BEATSYNC_GLOBALS.userPlaylist.Contains(MainActivity.GetKeyByValue(BEATSYNC_GLOBALS.songTitles, songList.GetItemAtPosition(nSongIndex).ToString())));
            }
          }
        }

        protected override void OnCreate(Bundle savedInstanceState)
        {
          // Default creation procedure
          base.OnCreate(savedInstanceState);

          // Set view to "Playlist" activity
          SetContentView(Resource.Layout.Playlist);

          // Initialize views
          bpmProgressBar  = FindViewById<ProgressBar>(Resource.Id.pbProgressBar);
          btnBack         = FindViewById<ImageView>(Resource.Id.imgBack);

          // Set-up the song list
          songList = FindViewById<ListView>(Resource.Id.lvSongs);
          songList.ChoiceMode = ChoiceMode.Multiple;

          // Change the color of the title text
          FindViewById<TextView>(Resource.Id.txtPlaylist).SetTextColor(BEATSYNC_CONSTANTS.Colors.TextGray);

          // Finish activity on "BACK" button click
          btnBack.Click += delegate
          {
            btnBack.Alpha = 0.55f;
```

```
                        Finish();
                    };

                    // Add or remove a song from the playlist on song click
                    songList.ItemClick += (sender, e) =>
                    {
                        // If item is now checked, add song to playlist
                        if (songList.IsItemChecked(e.Position))
                        {
                            BEATSYNC_GLOBALS.userPlaylist.Add(MainActivity.GetKeyByValue(BEATSYNC_GLOBALS.songTitles, songList.GetItemAtPosition(e.Position).ToString()));
                        }

                        // If item is now unchecked
                        else
                        {
                            // If playlist has enough songs to go on, remove song from playlist
                            if (BEATSYNC_GLOBALS.userPlaylist.Count > BEATSYNC_CONSTANTS.BEATSYNC_MINPLAYLIST)
                            {
                                BEATSYNC_GLOBALS.userPlaylist.Remove(MainActivity.GetKeyByValue(BEATSYNC_GLOBALS.songTitles, songList.GetItemAtPosition(e.Position).ToString()));
                            }

                            // Otherwise check the song back
                            else
                            {
                                songList.SetItemChecked(e.Position, true);
                                Toast.MakeText(this, "At least " + BEATSYNC_CONSTANTS.BEATSYNC_MINPLAYLIST + " songs are required.", ToastLength.Short).Show();
                            }
                        }

                        // Save altered playlist to disk
                        SavePlaylist();
                    };

                    // Preview a long-clicked song
                    songList.ItemLongClick += (sender, e) =>
                    {
                        // If program is passive
                        if (!BEATSYNC_GLOBALS.bActive)
                        {
                            // Preview chosen song
                            SetSong(MainActivity.GetKeyByValue(BEATSYNC_GLOBALS.songTitles, songList.GetItemAtPosition(e.Position).ToString()));
                        }
                    };

                    // If songs were successfully analyzed, refresh the song list
                    if (BEATSYNC_GLOBALS.bSongsAnalyzed == true)
                    {
                        RefreshSongList();
                        bpmProgressBar.Visibility = ViewStates.Gone;
                    }

                    // Change status bar color
                    if (Build.VERSION.SdkInt >= BuildVersionCodes.Lollipop)
                    {
                        Window.ClearFlags(WindowManagerFlags.TranslucentStatus);
                        Window.AddFlags(WindowManagerFlags.DrawsSystemBarBackgrounds);
                        Window.SetStatusBarColor(Color.DimGray);
                    }

                    // If program is passive
                    if (!BEATSYNC_GLOBALS.bActive)
                    {
                        // Notify user of preview feature
                        Toast.MakeText(this, "Hold to preview.", ToastLength.Long).Show();
                    }
                }
            }
        }
```

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:minWidth="25px"
    android:minHeight="25px">
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="35dp"
        android:layout_marginTop="10dp"
        android:layout_marginBottom="10dp"
        android:layout_marginLeft="10dp">
        <ImageView
            android:src="@drawable/icon_back"
            android:layout_height="match_parent"
            android:layout_width="30dp"
            android:id="@+id/imgBack" />
        <TextView
            android:text="Playlist"
            android:layout_height="match_parent"
            android:layout_width="wrap_content"
            android:layout_centerHorizontal="true"
            android:textSize="25sp"
            android:id="@+id/txtPlaylist" />
    </RelativeLayout>
    <ListView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/lvSongs" />
    <ProgressBar
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:indeterminate="true"
        android:id="@+id/pbProgressBar" />
</LinearLayout>
```

Settings.cs – BeatSync – קוד הפרוייקט

```
using System.IO;
using Android.App;
using Android.OS;
using Android.Views;
using Android.Widget;
using Android.Graphics;

namespace BeatSync
{
    [Activity(Label = "Settings")]
    public class Settings : Activity
    {
        // Current activity views
        private static NumberPicker agePicker;
        private static TextView     btnClearCache;
        private static TextView     btnDone;
        private static TextView     txtRange;

        /// <summary>
        /// Updates user age with the current number picker value.
        /// Updates the safe BPM range based on user's age.
        /// </summary>
        private static void UpdateAge()
        {
            int[] safeRange = MainActivity.SetAge(agePicker.Value);
            txtRange.Text = "Safe BPM range: " + safeRange[0] + " - " + safeRange[1];
        }

        protected override void OnCreate(Bundle savedInstanceState)
        {
            // Default creation procedure
            base.OnCreate(savedInstanceState);

            // Set view to "Settings" activity
            SetContentView(Resource.Layout.Settings);

            // Initialize range textview
            txtRange = FindViewById<TextView>(Resource.Id.txtRange);

            // Set-up the age number picker
            agePicker = FindViewById<NumberPicker>(Resource.Id.npAgePicker);
            agePicker.MinValue = BEATSYNC_CONSTANTS.BEATSYNC_MINIMUMAGE;
            agePicker.MaxValue = BEATSYNC_CONSTANTS.BEATSYNC_MAXIMUMAGE;
            agePicker.WrapSelectorWheel = false;
            agePicker.DescendantFocusability = DescendantFocusability.BlockDescendants;


            // If user age is stored, load it
            if (new Java.IO.File(Application.Context.FilesDir, BEATSYNC_CONSTANTS.BEATS
YNC_USERDATA).Exists())
            {
                using (var streamReader = new StreamReader(System.IO.Path.Combine(Files
Dir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERDATA)))
                {
                    agePicker.Value = (char)streamReader.Read();
                }
            }

            // Otherwise set to default
            else
            {
                agePicker.Value = BEATSYNC_CONSTANTS.BEATSYNC_DEFAULTAGE;
            }

            // Set-up the "CLEAR LOCAL CACHE" button
            btnClearCache = FindViewById<TextView>(Resource.Id.btnClearCache);
            btnClearCache.SetBackgroundColor(Color.WhiteSmoke);

            // Set-up the "DONE" button
            btnDone = FindViewById<TextView>(Resource.Id.btnDone);
            btnDone.SetTextColor(Color.White);
            btnDone.SetBackgroundColor(BEATSYNC_CONSTANTS.Colors.Blue);
            btnDone.Alpha = 0.85f;

            // Finish activity on "DONE" button click
            btnDone.Click += delegate
            {
                btnDone.Alpha = 0.75f;
                Finish();
            };

            // Clear cache and restart the app
            btnClearCache.LongClick += delegate
            {
                // Delete age
                if (new Java.IO.File(FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERDATA).Exists())
                {
                    new Java.IO.File(FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERDATA).Delete();
                }

                // Delete calculated songs list
                if (new Java.IO.File(FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERSONGS).Exists())
                {
                    new Java.IO.File(FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERSONGS).Delete();
                }

                // Delete playlist info
                if (new Java.IO.File(FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERPLAYLIST).Exists())
                {
                    new Java.IO.File(FilesDir, BEATSYNC_CONSTANTS.BEATSYNC_USERPLAYLIST).Delete();
                }

                    // Restart the application
                    Process.KillProcess(Process.MyPid());
                };

                // Update user age with the new number picker value
                agePicker.ValueChanged += delegate
                {
                    // Store age on disk
                    using (var streamWriter = new StreamWriter(System.IO.Path.Combine(FilesDir.AbsolutePath, BEATSYNC_CONSTANTS.BEATSYNC_USERDATA)))
                    {
                        streamWriter.Write((char)agePicker.Value);
                    }

                    // Update user age
                    UpdateAge();
                };

                // Update current user age
                UpdateAge();

                // Change status bar color
                if (Build.VERSION.SdkInt >= BuildVersionCodes.Lollipop)
                {
                    Window.ClearFlags(WindowManagerFlags.TranslucentStatus);
                    Window.AddFlags(WindowManagerFlags.DrawsSystemBarBackgrounds);
                    Window.SetStatusBarColor(BEATSYNC_CONSTANTS.Colors.Blue);
                }
            }
        }
    }
```

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:p1="http://schemas.android.com/apk/res/android"
    p1:orientation="vertical"
    p1:minWidth="25px"
    p1:minHeight="25px"
    p1:layout_width="match_parent"
    p1:layout_height="match_parent">
    <TextView
        p1:text="Pick your age"
        p1:layout_width="match_parent"
        p1:layout_height="wrap_content"
        p1:gravity="center"
        p1:layout_marginLeft="10dp"
        p1:layout_marginRight="10dp"
        p1:layout_marginTop="25dp"
        p1:textSize="22sp" />
    <NumberPicker
        p1:layout_width="match_parent"
        p1:layout_height="wrap_content"
        p1:id="@+id/npAgePicker"
        p1:minWidth="25px"
        p1:minHeight="25px" />
    <RelativeLayout
        p1:minWidth="25px"
        p1:minHeight="25px"
        p1:layout_width="match_parent"
        p1:layout_height="wrap_content">
        <TextView
            p1:text="Safe BPM range: "
            p1:layout_width="match_parent"
            p1:layout_height="wrap_content"
            p1:layout_margin="15dp"
            p1:gravity="center"
            p1:layout_centerHorizontal="true"
            p1:layout_alignParentTop="true"
            p1:textSize="18sp"
            p1:id="@+id/txtRange" />
        <TextView
            p1:text="CLEAR LOCAL CACHE"
            p1:layout_width="match_parent"
            p1:layout_height="wrap_content"
            p1:minHeight="50dp"
            p1:gravity="center"
            p1:layout_centerHorizontal="true"
            p1:layout_above="@+id/btnDone"
            p1:textSize="30sp"
            p1:id="@+id/btnClearCache" />
        <TextView
            p1:text="DONE"
            p1:layout_width="match_parent"
            p1:layout_height="wrap_content"
            p1:minHeight="50dp"
            p1:gravity="center"
            p1:layout_centerHorizontal="true"
            p1:layout_alignParentBottom="true"
            p1:textSize="30sp"
            p1:id="@+id/btnDone" />
    </RelativeLayout>  </LinearLayout>
```

BEATSYNC_VARIABLES.cs – BeatSync – קוד הפרוייקט

```
using System.Collections.Generic;
using Android.Runtime;
using Android.Graphics;
using Android.Media;
using Java.Util;

namespace BeatSync
{
    /// <summary>
    /// Application globals
    /// </summary>
    class BEATSYNC_GLOBALS
    {
        // Music related variables
        public static Dictionary<string, int>       songBPMs            = new Dictionary<string, int>();
        public static Dictionary<string, string>    songTitles          = new Dictionary<string, string>();
        public static JavaList<string>              userPlaylist        = new JavaList<string>();
        public static MediaPlayer                   currentSong         = new MediaPlayer();
        public static string                        currentlyPlaying    = null;
        public static bool                          bCrossfading        = false;
        public static bool                          bSongsAnalyzed      = false;

        // Application status variables
        public static bool bPaired = false;
        public static bool bActive = false;
    }

    /// <summary>
    /// Application constants.
    /// </summary>
    class BEATSYNC_CONSTANTS
    {
        /// <summary>
        /// Colors designed for BeatSync application.
        /// </summary>
        public class Colors
        {
            public static Color Gray        { get; } = Color.ParseColor("#7f8c8d");
            public static Color Red         { get; } = Color.ParseColor("#800000");
            public static Color Blue        { get; } = Color.ParseColor("#3498db");
            public static Color TextGray    { get; } = Color.DimGray;
        }

        // Application constants
        public static readonly string   BEATSYNC_APPLICATION_NAME   = "BeatSync";
        public static readonly int      BEATSYNC_NOTIFICATION_ID    = 7;
        public static readonly int      BEATSYNC_DEFAULTAGE         = 20;
        public static readonly int      BEATSYNC_MINIMUMAGE         = 16;
        public static readonly int      BEATSYNC_MAXIMUMAGE         = 79;
        public static readonly int      BEATSYNC_MAXBPM             = 220;
        public static readonly int      BEATSYNC_MINPLAYLIST        = 2;
        public static readonly float    BEATSYNC_SONGCUE            = 5.5f;
        public static readonly string   BEATSYNC_USERDATA           = "userdata.dat";
        public static readonly string   BEATSYNC_USERSONGS          = "usersongs.dat";
        public static readonly string   BEATSYNC_USERPLAYLIST       = "userplaylist.dat
";
        public static readonly int      REQUEST_ENABLE_BLUETOOTH    = 1;
        public static readonly string   ARDUINO_ADDRESS             = "98:D3:32:31:17:C
F";
        public static readonly string   ARDUINO_PAIRING_PIN         = "1234";
        public static readonly UUID     ARDUINO_UUID                = UUID.FromString("
00001101-0000-1000-8000-00805f9b34fb");
    }
}
```

```
using System;
using System.Text;
using System.IO;
using System.Linq;
using System.Collections.Generic;
using System.Numerics;

namespace BeatSync
{
    /// <summary>
    /// Resources used by the codec to encode/decode bitstreams.
    /// Resources include various constants and helping functions.
    /// </summary>
    class Resources
    {
        public class FourierTransform
        {
            /// <summary>
            /// Computes a Discrete Fourier Transform over given signal samples.
            /// Algorithm solves DFT formula over each sample:
            ///
            /// F(k) = 1 / Sqrt(N) * sigma(T(n) * e ^ (-2 * PI * n * k / N))
            ///
            /// Where:
            /// F(k)    - frequency bin of a sample;
            /// T(n)    - current sample;
            /// n       - sample index;
            /// k       - frequency index;
            /// N       - amount of samples in a given signal.
            ///
            /// Complexity: (n^2)
            ///
            /// </summary>
            /// <param name="Signal">Array of amplitude samples of a signal.</param>
            /// <returns>Array of frequency bins of a signal.</returns>
            public static Complex[] DFT(Complex[] Signal)
            {
                // Define formula variables
                int N = Signal.Length, n, k;
                Complex[] Spectrum = new Complex[N];

                // Iterate over signal samples
                for (n = 0;
                    n < N;
                    n++)
                {
                    // Define frequency sigma variable
                    Complex cpxFrequencySigma = new Complex(0, 0);

                    // Calculate frequency sigma over each sample, per sample
                    for (k = 0;
                        k < N;
                        k++)
                    {
                        cpxFrequencySigma += Signal[k] * Complex.Exp(new Complex(0, -
2 *
                                             Math.PI * n * k / N));
                    }

                    // Store calculated result
                    Spectrum[n] = (float)1 / Math.Sqrt(N) * cpxFrequencySigma;
                }

                // Return the spectrum
                return (Spectrum);
            }

            /// <summary>
            /// More efficient implementation of DFT.
            /// THIS CODE IS AN OPEN-SOURCE IMPLEMENTATION OF FFT AND WAS
            /// NOT WRITTEN FOR BEATSYNC.
            /// </summary>
            /// <param name="dir">FFT direction. 1 for FFT, -1 for Inverse FFT.</param>
            /// <param name="m">Power of 2 that is the sample amount.</param>
            /// <param name="x">Real part of the samples.</param>
            /// <param name="y">Imaginary part of the samples.</param>
            public static void FFT(short dir, int m, double[] x, double[] y)
            {
                int n, i, i1, j, k, i2, l, l1, l2;
                double c1, c2, tx, ty, t1, t2, u1, u2, z;

                // Calculate the number of points

                n = 1;

                for (i = 0; i < m; i++)
                    n *= 2;

                // Do the bit reversal

                i2 = n >> 1;
                j = 0;
                for (i = 0; i < n - 1; i++)
                {
                    if (i < j)
                    {
                        tx = x[i];
                        ty = y[i];
                        x[i] = x[j];
                        y[i] = y[j];
                               x[j] = tx;
                               y[j] = ty;
                           }
                           k = i2;

                           while (k <= j)
                           {
                               j -= k;
                               k >>= 1;
                           }

                           j += k;
                       }

                       // Compute the FFT

                       c1 = -1.0;
                       c2 = 0.0;
                       l2 = 1;

                       for (l = 0; l < m; l++)
                       {
                           l1 = l2;
                           l2 <<= 1;
                           u1 = 1.0;
                           u2 = 0.0;

                           for (j = 0; j < l1; j++)
                           {
                               for (i = j; i < n; i += l2)
                               {
                                   i1 = i + l1;
                                   t1 = u1 * x[i1] - u2 * y[i1];
                                   t2 = u1 * y[i1] + u2 * x[i1];
                                   x[i1] = x[i] - t1;
                                   y[i1] = y[i] - t2;
                                   x[i] += t1;
                                   y[i] += t2;
                               }

                               z = u1 * c1 - u2 * c2;
                               u2 = u1 * c2 + u2 * c1;
                               u1 = z;
                           }

                           c2 = Math.Sqrt((1.0 - c1) / 2.0);

                           if (dir == 1)
                               c2 = -c2;

                           c1 = Math.Sqrt((1.0 + c1) / 2.0);
                       }

                       // Scaling for forward transform

                       if (dir == 1)
                       {
                           for (i = 0; i < n; i++)
                           {
                               x[i] /= n;
                               y[i] /= n;
                           }
                       }
                   }

                   /// <summary>
                   /// Computes an Inverse Discrete Fourier Transform over given signal samples.
                   /// Algorithm solves Inverse DFT formula over each sample:
                   ///
                   /// T(n) = 1 / Sqrt(N) * sigma(F(k) * e ^ (2 * PI * n * k / N))
                   ///
                   /// Where:
                   /// F(k)    - frequency bin of a sample;
                   /// T(n)    - current sample;
                   /// n       - sample index;
                   /// k       - frequency index;
                   /// N       - amount of samples in a given signal.
                   ///
```

```
            /// Complexity: (n^2)
            ///
            /// </summary>
            /// <param name="Spectrum">Array of frequency bins of a signal.</param>
            /// <returns>Array of amplitude samples of a signal.</returns>
            public static Complex[] InverseDFT(Complex[] Spectrum)
            {
                // Define formula variables
                int N = Spectrum.Length, n, k;
                Complex[] Signal = new Complex[N];

                // Iterate over frequency bins
                for (n = 0;
                    n < N;
                    n++)
                {
                    // Define sample sigma variable
                    Complex cpxSample = new Complex(0, 0);

                    // Calculate sigma
                    for (k = 0;
                        k < N;
                        k++)
                    {
                        cpxSample += Spectrum[k] * Complex.Exp(new Complex(0, 2 * Math.PI * n * k / N));
                    }

                    // Store calculated result
                    Signal[n] = (float)1 / Math.Sqrt(N) * cpxSample;
                }

                // Return the signal
                return (Signal);
            }

        }

        // Bit-depths supported by this codec
        public static readonly int[] ALLOWED_BITDEPTHS =
        {
            sizeof(short) * 8,
            sizeof(float) * 8,
        };

        // Max amplitude for each supported bit-depth
        public static readonly Dictionary<Type, int> MAX_AMPLITUDE  = new Dictionary<Type, int>
        {
            { typeof(short[]),    short.MaxValue    },
            { typeof(float[]),    1                 }
        };
        public const int GROUP_ID_LENGTH = 4;

        /// <summary>
        /// Checks if given file path is valid for reading and writing.
        /// </summary>
        /// <param name="sFilepath">Path to a file.</param>
        /// <returns>True if file path is valid, False otherwise.</returns>
        public static bool IsValidFilepath(string sFilepath)
        {
            // Try writing a dummy file to file path
            try
            {
                if (!File.Exists(sFilepath))
                {
                    File.WriteAllText(sFilepath, "");
                    File.Delete(sFilepath);
                }

                return (true);
            }

            // If file could not be written, file path is invalid
            catch
            {
                return (false);
            }
        }

        /// <summary>
        /// Returns index of the first occurence of target array in source array
        /// ranging from the start index to search limit index. If subarray was not found,
        /// returns -1.
        /// </summary>
        /// <param name="arrbSource">Byte array to look for subarray in.</param>
        /// <param name="arrbTarget">Byte array to look for in the source array.</param>
        /// <param name="nStartIndex">Index to begin searching from in the source array.</param>
        /// <param name="nSearchLimit">Amount of bytes to scan in the source array.</param>
        /// <returns>
        /// Index of the first occurence of target array in source array.
        /// If subarray was not found, returns -1.
        /// </returns>
        public static int IndexOf(byte[] arrbSource,
                                  byte[] arrbTarget,
                                  int nStartIndex,
                                  int nSearchLimit)
        {
            // If passed values are within limits
            if (arrbSource.Length >= arrbTarget.Length &&
                arrbTarget.Length > 0 &&
                arrbSource.Length - nStartIndex > nSearchLimit &&
                arrbSource.Length > nStartIndex)
            {
                // Iterate over source array from the given index
                for (int nIndexSource = nStartIndex;
                    nIndexSource < nStartIndex + nSearchLimit;
                    nIndexSource++)
                {
                    // If the first byte of target array was found in the source array
                    if (arrbSource[nIndexSource] == arrbTarget[0])
                    {
                        // Define a 'found' flag boolean as true
                        bool bFound = true;
```

```

                            // Iterate over target array
                            for (int nIndexTarget = 1;
                                nIndexTarget < arrbTarget.Length && bFound == true;
                                nIndexTarget++)
                            {
                                // If bytes do not match, mark 'found' flag as false
                                if (arrbSource[nIndexSource + nIndexTarget] != arrbTarget[nIndexTarget])
                                {
                                    bFound = false;
                                }
                            }

                            // If found flag is true after the iteration, subarray was found
                            if (bFound == true)
                            {
                                return (nIndexSource);
                            }
                        }
                    }
                }

                // If subarray was not found, return an illegal index value
                return (-1);
            }
        }

        /// <summary>
        /// Header information struct.
        /// </summary>
        public struct WAV_Header
        {
            public char[]   sGroupID;       // Chunk ID (should be 'RIFF')
            public uint     dwFileLength;   // File size - 8 (without Group ID and RIFF type)
            public char[]   sRiffType;      // Extension of a RIFF file (should be 'WAVE')
        }

        /// <summary>
        /// Format chunk information struct.
        /// </summary>
        public struct WAV_FormatChunk
        {
            public char[]   sGroupID;                   // Chunk ID (should be 'fmt ')
            public uint     dwChunkSize;                // Size of the rest of the chunk which follows this number
            public ushort   wFormatTag;                 // Sample format (should be 1 for 'PCM')
            public ushort   wChannels;                  // Amount of audio channels present
            public uint     dwSampleRate;               // Amount of samples per second of audio
            public uint     dwAverageBytesPerSecond;    // Average amount of bytes per second audio
            public ushort   wBlockAlign;                // Number of audio channels * Bits per Sample / 8
            public ushort   wBitDepth;                  // Amount of bits per audio sample
        }

        /// <summary>
        /// Data chunk information struct.
        /// </summary>
        public struct WAV_DataChunk
        {
            public char[]   sGroupID;       // Chunk ID (should be 'data')
            public uint     dwChunkSize;    // Number of bytes in the sample data portion
            public Array    sampleData;     // Array of audio samples
        }

        /// <summary>
        /// WAV file extension container.
        /// Class can decode valid WAV files, make changes to them and encode to disk.
        /// </summary>
        class WAV
        {
            // Audio container information variables
            private static WAV_Header       hHeader;
            private static WAV_FormatChunk  fcFormat;
            private static WAV_DataChunk    dcData;
            private static string           sFilePath = null;
            private static float            fFileDuration;
            private static long             lFileSize;
            private static int              nFileBPM;

            // Other related variables
            private static readonly int BPM_WINDOW_SIZE = 8;
            private static readonly int MIN_FILE_LENGTH = 36;

            /// <summary>
            /// Path to loaded audio file.
            /// </summary>
            public string Path
            {
                get
                {
                    return sFilePath;
                }
            }

            /// <summary>
            /// Header chunk struct of audio.
            /// </summary>
            public WAV_Header Header
            {
                get
                {
                    return hHeader;
                }
            }

            /// <summary>
            /// Format chunk struct of audio.
            /// </summary>
            public WAV_FormatChunk Format
            {
                get
                {
                    return fcFormat;
                }
            }

            /// <summary>
            /// Data chunk struct of audio.
            /// </summary>
            public WAV_DataChunk Data
            {
                get
                {
                    return dcData;
                }
            }

            /// <summary>
            /// Duration of audio in seconds.
            /// </summary>
            public double Duration
            {
                get
                {
                    return (IsLoaded() ? Math.Round(fFileDuration, 3) : -1);
                }
            }

            /// <summary>
            /// Minimum audio length (in seconds) for BPM calculation.
            /// </summary>
            public static double BPM_CALCULATION_DURATION_MINIMUM
            {
                get
                {
                    return BPM_WINDOW_SIZE * 3;
                }
            }

            /// <summary>
            /// Size of audio file in bytes.
            /// </summary>
            public long Size
            {
                get
                {
                    return (IsLoaded() ? lFileSize : -1);
                }
            }

            /// <summary>
            /// Amount of samples per second of audio.
            /// </summary>
            public int SampleRate
            {
                get
                {
                    return (IsLoaded() ? (int)fcFormat.dwSampleRate : -1);
                }
            }

            /// <summary>
            /// Amount of bits per audio sample.
            /// </summary>
            public int BitDepth
            {
                get
                {
                    return (IsLoaded() ? fcFormat.wBitDepth : -1);
                }
            }

            /// <summary>
            /// Amount of Beats Per Minute in a given audio file.
            /// </summary>
            public int BPM
            {
                get
                {
                    // Set window time to be 8 seconds
                    int nWindow = BPM_WINDOW_SIZE;

                    // If file is loaded, BPM was not yet calculated and song length is long enough
                    if (IsLoaded() &&
                        nFileBPM == -1 &&
                        Duration >= nWindow * 3)
                    {
                        // Calculate the amount of samples needed for 8 seconds of audio
                        int nSamples = SampleRate * nWindow;

                        // Create an array of window sample summations
                        double[] Peaks = new double[((int)Duration / 3) / nWindow];

                        // Define a separate sample array
                        Array AudioSignal = null;

                        // Copy samples to new sample array and normalize them
                        switch (Format.wBitDepth / 8)
                        {
                            case (sizeof(short)):
                                AudioSignal = Array.CreateInstance(typeof(short), Data.sampleData.Length);
                                Array.Copy(Data.sampleData, AudioSignal, AudioSignal.Length);
                                Samples.Normalize((short[])AudioSignal);
                                break;
                            case (sizeof(float)):
                                AudioSignal = Array.CreateInstance(typeof(float), Data.sampleData.Length);
                                Array.Copy(Data.sampleData, AudioSignal, AudioSignal.Length);
                                Samples.Normalize((float[])AudioSignal);
                                break;
                        }

                        // Go over the middle third of the audio data
```

```
for (int nWindowIndex = 0, nSampleIndex = AudioSignal.Length / 3;
    nWindowIndex < Peaks.Length;
    nWindowIndex++)
{
    // Go over the current window
    for (int nWindowEnd = nSampleIndex + nSamples;
        nSampleIndex < nWindowEnd;
        nSampleIndex++)
    {
        // Count the amount of 0db peaks in the current window
        switch (Format.wBitDepth / 8)
        {
            case (sizeof(short)):
                Peaks[nWindowIndex] += Math.Abs((short)AudioSignal.GetValue(nSampleIndex) - 1) == Resources.MAX_AMPLITUDE[typeof(short[])] - 1 ? 1 : 0;
                break;
            case (sizeof(float)):
                Peaks[nWindowIndex] += Math.Abs((float)AudioSignal.GetValue(nSampleIndex) - 1) == Resources.MAX_AMPLITUDE[typeof(float[])] - 1 ? 1 : 0;
                break;
        }
    }
}

// Get the index of the first sample of the "loudest" window
int nLeadSample = (Peaks.ToList().IndexOf(Peaks.Max()) * nWindow) + (Peaks.Length * nWindow) * SampleRate;

// Define variables for Fast Fourier Transform
int N = nSamples;
int nPower = 0;
int nIndex;

// Find a power of 2 that is the closest to the amount of samples in a window
for (nIndex = 1;
    (nIndex * 2) < N;
    nIndex *= 2, nPower++) ;

// Set the amount of samples to be 2 in power we just found
N = (int)Math.Pow(2, nPower);

// Define arrays for real and imaginary parts of the signal
double[] Real       = new double[N];
double[] Imaginary  = new double[N];

// Copy the amplitude values from audio data to the real part of the signal
switch (Format.wBitDepth / 8)
{
    case (sizeof(short)):
        Samples.LoadSamples(Real, (short[])AudioSignal, nLeadSample, N);
        break;
```

```
                                case (sizeof(float)):
                                    Samples.LoadSamples(Real, (float[])AudioSignal, nLeadSample, N);
                                    break;
                            }

                            // Calculate a Fast Fourier Transform over given signal
                            Resources.FourierTransform.FFT(1, nPower, Real, Imaginary);

                            // Calculate a spectrum coefficient
                            float nSpectrumCoefficient = (SampleRate / (float)N) * 2;
                            int nSample;

                            // Reach the 250hz frequency bin
                            for (nSample = 0;
                                nSample * nSpectrumCoefficient < 250;
                                nSample++) ;

                            // Set every frequency bin beginning from 100hz to be 0
                            for (;
                                nSample < N;
                                nSample++)
                            {
                                Real[nSample] = Imaginary[nSample] = 0;
                            }

                            // Calculate an Inverse Fast Fourier Transform over given signal
                            Resources.FourierTransform.FFT(-1, nPower, Real, Imaginary);

                            // Define variables for lowpassed signal analysis
                            float[] Lowpass = new float[N];
                            double dAverage = 0;
                            double dSummation = 0;

                            // Iterate over each sample in the real array
                            for (nIndex = 0; nIndex < N; nIndex++)
                            {
                                // Copy lowpassed signal to the lowpass array, setting each negative value to 0
                                switch (Format.wBitDepth / 8)
                                {
                                    case (sizeof(short)):
                                        Lowpass[nIndex] = Math.Max((short)0, (short)Real[nIndex]);
                                        break;
                                    case (sizeof(float)):
                                        Lowpass[nIndex] = Math.Max((float)0, (float)Real[nIndex]);
                                        break;
                                }
                            }

                            // Normalize the received lowpass signal
                            switch (Format.wBitDepth / 8)
                            {
                                case (sizeof(short)):
                                    // Convert the float array to short array
                                    Samples.Normalize(Array.ConvertAll(Lowpass, new Converter<float, short>
                                    (
                                        // Cast each float sample to short sample
                                        delegate (float fSample)
                                        {
                                            return (short)fSample;
                                        }
                                    )));
                                    break;
                                case (sizeof(float)):
                                    Samples.Normalize(Lowpass);
                                    break;
                            }

                            // Create a list of pairs for impulse matches
                            List<KeyValuePair<int, int>> Matches;

                            // Summate lowpassed signal amplitudes
                            for (nIndex = 0; nIndex < N; nIndex++)
                            {
                                dSummation += Lowpass[nIndex];
                            }

                            // Set the average multiplier to start from 7
                            int nMultiplier = 7;

                            // Start looking for bass drum impulses in the lowpassed signal.
                            // Amplitude threshold is calculated by taking the average amplitude value
                            // and multiplying it by the multiplier variable.
                            // If fewer than two matches were found above the threshold, lower the
                            // threshold value by decrementing the multiplier value and try again.
                            do
                            {
                                // Calculate average sample amplitude times the average multiplier
                                dAverage = dSummation / N * nMultiplier--;

                                // Initialize a new list of matches
                                Matches = new List<KeyValuePair<int, int>>();

                                // Calculate the jumping distance from an impulse
                                int nDistance = SampleRate / 4;

                                // Iterate over lowpass signal
                                for (nIndex = 0; nIndex < N; nIndex++)
                                {
                                    // If an amplitude above the average value was found

                                    if (Lowpass[nIndex] > dAverage)
                                    {
                                        // Find the next impulse
                                        for (int nImpulse = nIndex + nDistance; nImpulse < N; nImpulse++)
                                        {
                                            // If another impulse was found
                                            if (Lowpass[nImpulse] > dAverage)
                                            {
                                                // Make a match out of the first impulse index and
                                                // delta of two found impulses
                                                Matches.Add(new KeyValuePair<int, int>(nIndex, nImpulse - nIndex));
                                                nImpulse += nDistance;
                                            }
                                        }

                                        // Jump forward from the current impulse
                                        nIndex += nDistance;
                                    }
                                }
                            }
                            while (Matches.Count < 2 || nMultiplier < 0);

                            // If at least two matches exist
                            if (nMultiplier != 0)
                            {
                                // Define variables for the answer pair and jump counter
                                KeyValuePair<int, int> Answer = new KeyValuePair<int, int>(0, 0);
                                int nJumps = 0;

                                // Iterate over each impulse in the list of matches
                                foreach (KeyValuePair<int, int> Impulse in Matches)
                                {
                                    // Start from the second impulse index
                                    // Add delta from the pair to jump to the presumed index of the next impulse
                                    for (nIndex = Impulse.Key + Impulse.Value; nIndex < N; nIndex += Impulse.Value)
                                    {
                                        // If jump landed on another impulse
                                        if (Lowpass[nIndex] > dAverage)
                                        {
                                            // Increment the jump counter
                                            nJumps++;
                                        }

                                        // Otherwise fulfill the loop condition
                                        else
                                        {
                                            nIndex = N;
                                        }
                                    }

                                    // If successful jumps were made
                                    if (nJumps > 0)
                                    {
                                        // Set the answer to be the current impulse delta if current answer's jump amount is lower
                                        Answer = nJumps > Answer.Key ? new KeyValuePair<int, int>(nJumps, Impulse.Value) : Answer;
                                    }

                                    // Set the jump counter back to 0;
                                    nJumps = 0;
                                }

                                // To calculate BPM using the sample delta, use formula:
                                // BPM = 60 / (Delta / Sample Rate)
                                nFileBPM = (int)(60 / ((float)Answer.Value / SampleRate));

                                // If BPM is lower than 100 or bigger than 200, scale it up or down appropriately
                                while (nFileBPM < 100 || nFileBPM > 200)
                                {
                                    nFileBPM = nFileBPM < 100 ? nFileBPM * 2 : nFileBPM > 200 ? nFileBPM / 2 : nFileBPM;
                                }
                            }
                        }

                        return (nFileBPM);
                    }
                }

                /// <summary>
                /// A set of functions that work with uncompressed audio samples.
                /// </summary>
                class Samples
                {
                    /// <summary>
                    /// Loads samples from byte array to short array.
                    /// </summary>
                    public static void LoadSamples(short[] Destination, byte[] Source, int nSampleDataIndex)
                    {
                        // Calculate sample size in bytes
                        int nSampleSize = sizeof(short);

                        // Load destination array with converted samples
                        for (int nSampleIndex = 0;
                            nSampleIndex < Destination.Length;
                            nSampleIndex++)
                        {
                            Destination[nSampleIndex] = BitConverter.ToInt16(Source, nSampleDataIndex + (nSampleIndex * nSampleSize));
                        }
                    }

                    /// <summary>
                    /// Loads samples from byte array to float array.
                    /// </summary>
                    public static void LoadSamples(float[] Destination, byte[] Source, int nSampleDataIndex)
                    {
                        // Calculate sample size in bytes
                        int nSampleSize = sizeof(float);

                        // Load destination array with converted samples
                        for (int nSampleIndex = 0;
                            nSampleIndex < Destination.Length;
                            nSampleIndex++)
                        {
                            Destination[nSampleIndex] = BitConverter.ToSingle(Source, nSampleDataIndex + (nSampleIndex * nSampleSize));
                        }
                    }

                    /// <summary>
                    /// Loads samples from short array to double array.
                    /// </summary>
                    public static void LoadSamples(double[] Destination, short[] Source, int nSampleDataIndex, int nSampleAmount)
                    {
                        // Load destination array with source samples
                        for (int nSampleIndex = 0;
                            nSampleIndex < nSampleAmount;
                            nSampleIndex++)
                        {
                            Destination[nSampleIndex] = Source[nSampleDataIndex + nSampleIndex];
                        }
                    }

                    /// <summary>
                    /// Loads samples from float array to double array.
                    /// </summary>
                    public static void LoadSamples(double[] Destination, float[] Source, int nSampleDataIndex, int nSampleAmount)
                    {
                        // Load destination array with source samples
                        for (int nSampleIndex = 0;
                            nSampleIndex < nSampleAmount;
                            nSampleIndex++)
                        {
                            Destination[nSampleIndex] = Source[nSampleDataIndex + nSampleIndex];
                        }
                    }

                    /// <summary>
                    /// Normalized a signal of short samples.
                    /// </summary>
                    /// <param name="Samples">Signal of short samples.</param>
                    public static void Normalize(short[] Samples)
                    {
                        // Define a max amplitude variable
                        int nMaxAmplitude = 1;

                        // Iterate over sample array and find highest amplitude value
                        for (int nSampleIndex = 0;
                            nSampleIndex < Samples.Length;
                            nSampleIndex++)
                        {
                            nMaxAmplitude = Math.Abs((int)Samples[nSampleIndex]) > nMaxAmplitude ?
                                            Math.Abs((int)Samples[nSampleIndex]) : nMaxAmplitude;
                        }

                        // Calculate sample multiplication coefficient
                        float fCoefficient = Resources.MAX_AMPLITUDE[typeof(short[])] / (float)nMaxAmplitude;

                        // If samples are not already normalized
                        if (fCoefficient > 1)
```

```
            {
                // Iterate over sample array and multiply each sample amplitude by coefficient
                for (int nSampleIndex = 0;
                    nSampleIndex < Samples.Length;
                    nSampleIndex++)
                {
                    Samples[nSampleIndex] = (short)(Samples[nSampleIndex] * fCoefficient);
                }
            }
        }

        /// <summary>
        /// Normalizes a signal of float samples.
        /// </summary>
        /// <param name="Samples">Signal of float samples.</param>
        public static void Normalize(float[] Samples)
        {
            // Define a max amplitude variable
            float fMaxAmplitude = 1;

            // Iterate over sample array and find highest amplitude value
            for (int nSampleIndex = 0;
                nSampleIndex < Samples.Length;
                nSampleIndex++)
            {
                fMaxAmplitude = Math.Abs(Samples[nSampleIndex]) > fMaxAmplitude ?
                                Math.Abs(Samples[nSampleIndex]) : fMaxAmplitude;
            }

            // Calculate sample multiplication coefficient
            float fCoefficient = Resources.MAX_AMPLITUDE[typeof(float[])] / fMaxAmplitude;

            // If samples are not already normalized
            if (fCoefficient != 1)
            {
                // Iterate over sample array and multiply each sample amplitude by coefficient
                for (int nSampleIndex = 0;
                    nSampleIndex < Samples.Length;
                    nSampleIndex++)
                {
                    Samples[nSampleIndex] *= fCoefficient;
                }
            }
        }

        /// <summary>
        /// Reverses polarity of short signal.
        /// </summary>
        /// <param name="Samples">Short signal.</param>
        public static void ReversePolarity(short[] Samples)
        {
            // Iterate over sample array and multiply each amplitude by -1
            for (int nSampleIndex = 0;
                nSampleIndex < Samples.Length;
                nSampleIndex++)
            {
                Samples[nSampleIndex] = (short)-Samples[nSampleIndex];
            }
        }

        /// <summary>
        /// Reverses polarity of float signal.
        /// </summary>
        /// <param name="Samples">Float signal.</param>
        public static void ReversePolarity(float[] Samples)
        {
            // Iterate over sample array and multiply each amplitude by -1
            for (int nSampleIndex = 0;
                nSampleIndex < Samples.Length;
                nSampleIndex++)
            {
                Samples[nSampleIndex] = -Samples[nSampleIndex];
            }
        }

        /// <summary>
        /// Reverses a signal of short samples.
        /// </summary>
        /// <param name="Samples">Signal of short samples.</param>
        public static void Reverse(short[] Samples)
        {
            // Define a temporary sample value
            short nSample;

            // Iterate over half of the sample array
            for (int nSampleIndex = 0;
                nSampleIndex < Samples.Length / 2;
                nSampleIndex++)
            {
                // Swap the given sample with a mirrored sample from the end

                nSample = Samples[Samples.Length - nSampleIndex - 1];
                Samples[Samples.Length - nSampleIndex - 1] = Samples[nSampleIndex];
                Samples[nSampleIndex] = nSample;
            }
        }

        /// <summary>
        /// Reverses a signal of float samples.
        /// </summary>
        /// <param name="Samples">Signal of float samples.</param>
        public static void Reverse(float[] Samples)
        {
            // Define a temporary sample value
            float nSample;

            // Iterate over half of the sample array
            for (int nSampleIndex = 0;
                nSampleIndex < Samples.Length / 2;
                nSampleIndex++)
            {
                // Swap the given sample with a mirrored sample from the end

                nSample = Samples[Samples.Length - nSampleIndex - 1];
```

```
                        Samples[Samples.Length - nSampleIndex - 1] = Samples[nSampleIndex];
                        Samples[nSampleIndex] = nSample;
                    }
                }

                /// <summary>
                /// Writes samples from short array to byte buffer.
                /// </summary>
                public static void WriteSamples(byte[] Buffer, short[] Samples, int nBufferIndex)
                {
                    // Calculate sample size
                    int nSampleSize = fcFormat.wBitDepth / 8;

                    // Iterate over each sample, convert it to bytes and write to buffer
                    for (int nSampleIndex = 0;
                        nSampleIndex < Samples.Length;
                        nSampleIndex++)
                    {
                        BitConverter.GetBytes(Samples[nSampleIndex]).CopyTo(Buffer, nBufferIndex + (nSampleIndex * nSampleSize));
                    }
                }

                /// <summary>
                /// Writes samples from float array to byte buffer.
                /// </summary>
                public static void WriteSamples(byte[] Buffer, float[] Samples, int nBufferIndex)
                {
                    // Calculate sample size
                    int nSampleSize = fcFormat.wBitDepth / 8;

                    // Iterate over each sample, convert it to bytes and write to buffer
                    for (int nSampleIndex = 0;
                        nSampleIndex < Samples.Length;
                        nSampleIndex++)
                    {
                        BitConverter.GetBytes(Samples[nSampleIndex]).CopyTo(Buffer, nBufferIndex + (nSampleIndex * nSampleSize));
                    }
                }
            }

            /// <summary>
            /// Builds a WAV container for a given WAVE file.
            /// </summary>
            /// <param name="sPathToFile">Path to an audio file.</param>
            public WAV(string sPathToFile)
            {
                Load(sPathToFile);
            }

            /// <summary>
            /// Builds a WAV container out of a given WAV container.
            /// </summary>
            /// <param name="waveFile">Given WAV container.</param>
            public WAV(WAV waveFile)
```

```
                {
                    if (waveFile.IsLoaded())
                    {
                        // Unload previous file information
                        this.Unload();

                        // Fill in WAV header information
                        hHeader = new WAV_Header
                        {
                            sGroupID        = waveFile.Header.sGroupID,
                            dwFileLength    = waveFile.Header.dwFileLength,
                            sRiffType       = waveFile.Header.sRiffType
                        };

                        // Fill in WAV format chunk information
                        fcFormat = new WAV_FormatChunk
                        {
                            sGroupID                = waveFile.Format.sGroupID,
                            dwChunkSize             = waveFile.Format.dwChunkSize,
                            wFormatTag              = waveFile.Format.wFormatTag,
                            wChannels               = waveFile.Format.wChannels,
                            dwSampleRate            = waveFile.Format.dwSampleRate,
                            dwAverageBytesPerSecond = waveFile.Format.dwAverageBytesPerSecond,
                            wBlockAlign             = waveFile.Format.wBlockAlign,
                            wBitDepth               = waveFile.Format.wBitDepth
                        };

                        // Fill in WAV data chunk information
                        dcData = new WAV_DataChunk
                        {
                            sGroupID    = waveFile.Data.sGroupID,
                            dwChunkSize = waveFile.Data.dwChunkSize
                        };

                        // Set file path to be a non-null value
                        sFilePath = "N/A";

                        // Decide upon the sample array type and load it with samples
                        switch (fcFormat.wBitDepth / 8)
                        {
                            case (sizeof(short)):
                                dcData.sampleData = Array.CreateInstance(typeof(short), (dcData.dwChunkSize * 8) / fcFormat.wBitDepth);
                                break;
                            case (sizeof(float)):
                                dcData.sampleData = Array.CreateInstance(typeof(float), (dcData.dwChunkSize * 8) / fcFormat.wBitDepth);
                                break;
                            default:
                                Unload();
                                break;
                        }

                        // If sample data was loaded successfully
                        if (IsLoaded())
                        {
                            // Copy audio samples to a new WAV
                            waveFile.Data.sampleData.CopyTo(dcData.sampleData, 0);

                            // Calculate new file size
                            lFileSize = System.Runtime.InteropServices.Marshal.SizeOf(hHeader) +             // Header size
                                        System.Runtime.InteropServices.Marshal.SizeOf(fcFormat) +            // Format chunk size
                                        (sizeof(byte) * Resources.GROUP_ID_LENGTH) +                         // 'Group ID' variable type size
                                        System.Runtime.InteropServices.Marshal.SizeOf(dcData.dwChunkSize) + // 'Chunk size' variable type size
                                        dcData.dwChunkSize;

                            // Set file path to be a non-null value
                            sFilePath = "N/A";

                            // WAV File Duration = Data Chunk Size / Average Bytes per Second
                            fFileDuration = (float)dcData.dwChunkSize / fcFormat.dwAverageBytesPerSecond;
                        }
                    }
                }

                /// <summary>
                /// Builds a WAV container out of sample array, sample rate, channel amount and bit-depth.
                /// </summary>
                /// <param name="Data">Audio samples.</param>
                /// <param name="nSampleRate">Sample rate.</param>
                /// <param name="nChannels">Amount of channels.</param>
                /// <param name="nBitDepth">Bit-depth.</param>
                public WAV(byte[] Data,
                           uint nSampleRate,
                           ushort nChannels,
                           ushort nBitDepth)
                {
                    // Unload previous file information
                    Unload();

                    // Fill in WAV header information
                    hHeader = new WAV_Header
                    {
                        sGroupID        = new char[] { 'R', 'I', 'F', 'F' },
                        dwFileLength    = 0,
                        sRiffType       = new char[] { 'W', 'A', 'V', 'E' }
                    };

                    // Fill in WAV format chunk information
                    fcFormat = new WAV_FormatChunk
                    {
                        sGroupID                = new char[] { 'f', 'm', 't', ' ' },
                        dwChunkSize             = 16,
                        wFormatTag              = 1,
                        wChannels               = nChannels,
                        dwSampleRate            = nSampleRate,
                        dwAverageBytesPerSecond = nSampleRate * nChannels * nBitDepth / 8,
                        wBlockAlign             = (ushort)(nChannels * nBitDepth / 8),
                        wBitDepth               = nBitDepth
                    };

                    // Fill in WAV data chunk information
                    dcData = new WAV_DataChunk
```

```csharp
            {
                sGroupID    = new char[] { 'd', 'a', 't', 'a' },
                dwChunkSize = (uint)Data.Length
            };

            // Set file path to be a non-null value
            sFilePath = "N/A";

            // Decide upon the sample array type and load it with samples
            switch (fcFormat.wBitDepth / 8)
            {
                case (sizeof(short)):
                    dcData.sampleData = Array.CreateInstance(typeof(short), (dcData.dwChunkSize * 8) / fcFormat.wBitDepth);
                    Samples.LoadSamples((short[])dcData.sampleData, Data, 0);
                    break;
                case (sizeof(float)):
                    dcData.sampleData = Array.CreateInstance(typeof(float), (dcData.dwChunkSize * 8) / fcFormat.wBitDepth);
                    Samples.LoadSamples((float[])dcData.sampleData, Data, 0);
                    break;
                default:
                    Unload();
                    break;
            }

            // If samples were loaded successfully
            if (IsLoaded())
            {
                // Calculate new file size
                lFileSize = System.Runtime.InteropServices.Marshal.SizeOf(hHeader) +            // Header size
                            System.Runtime.InteropServices.Marshal.SizeOf(fcFormat) +           // Format chunk size
                            (sizeof(byte) * Resources.GROUP_ID_LENGTH) +                        // 'Group ID' variable type size
                            System.Runtime.InteropServices.Marshal.SizeOf(dcData.dwChunkSize) + // 'Chunk size' variable type size
                            dcData.dwChunkSize;

                // WAV File Duration = Data Chunk Size / Average Bytes per Second
                fFileDuration = (float)dcData.dwChunkSize / fcFormat.dwAverageBytesPerSecond;
            }
        }

        /// <summary>
        /// Checks if given file is a WAV file.
        /// </summary>
        /// <param name="sPathToFile">Path to a file.</param>
        /// <returns>True if given file is a valid WAV file, False otherwise.</returns>
        public static bool IsWAV(string sPathToFile)
        {
            // If file exists and has a minimum amount of bytes
            if (File.Exists(sPathToFile) &&
                new FileInfo(sPathToFile).Length > MIN_FILE_LENGTH)
            {
                // Open a stream for a given file and read the needed amount of bytes
                FileStream fsStream = File.OpenRead(sPathToFile);
                byte[] arrbBuffer = new byte[MIN_FILE_LENGTH];
                fsStream.Read(arrbBuffer, 0, arrbBuffer.Length);

                // Check if:
                //  - File is a RIFF file
                //  - RIFF type is WAVE
                //  - This codec can work with this file's bit depth
                return (Encoding.ASCII.GetString(arrbBuffer, 0, 4).Equals("RIFF") &&
                        Encoding.ASCII.GetString(arrbBuffer, 8, 4).Equals("WAVE") &&
                        Resources.ALLOWED_BITDEPTHS.Contains(System.BitConverter.ToUInt16(arrbBuffer, 34)));
            }

            return (false);
        }

        /// <summary>
        /// Loads audio information into the given instance.
        /// </summary>
        /// <param name="sPathToFile">Path to an audio file (include extension).</param>
        /// <returns>True if file was loaded successfully, False otherwise.</returns>
        public bool Load(string sPathToFile)
        {
            // If file exists AND is a valid WAV file
            if (File.Exists(sPathToFile) &&
                IsWAV(sPathToFile))
            {
                // Read all bytes from a given WAV file
                FileStream fs = File.OpenRead(sPathToFile);
                byte[] arrbFileData = File.ReadAllBytes(sPathToFile);
                int nFileDataIndex;

                // Unload previous file information
                Unload();

                // Fill in WAV header information
                hHeader = new WAV_Header
                {
                    sGroupID        = Encoding.ASCII.GetChars(arrbFileData, (nFileDataIndex = 0), sizeof(byte) * Resources.GROUP_ID_LENGTH),
                    dwFileLength    = System.BitConverter.ToUInt32(arrbFileData, (nFileDataIndex += sizeof(byte) * Resources.GROUP_ID_LENGTH)),
                    sRiffType       = Encoding.ASCII.GetChars(arrbFileData, (nFileDataIndex += sizeof(System.Int32)), Resources.GROUP_ID_LENGTH)
                };

                // Find format chunk index
                if ((nFileDataIndex = Resources.IndexOf(arrbFileData,
                                                        Encoding.ASCII.GetBytes("fmt "),
                                                        (nFileDataIndex += sizeof(byte) * Resources.GROUP_ID_LENGTH),
                                                        System.Runtime.InteropServices.Marshal.SizeOf(fcFormat))) == -1)
                {
                    return (false);
                }

                // Fill in WAV format chunk information
                fcFormat = new WAV_FormatChunk
                {
                    sGroupID               = Encoding.ASCII.GetChars(arrbFileData,           nFileDataIndex, sizeof(byte) * Resources.GROUP_ID_LENGTH),
                    dwChunkSize            = System.BitConverter.ToUInt32(arrbFileData,    (nFileDataIndex += sizeof(byte) * Resources.GROUP_ID_LENGTH)),
                    wFormatTag             = System.BitConverter.ToUInt16(arrbFileData,    (nFileDataIndex += sizeof(System.Int32))),
                    wChannels              = System.BitConverter.ToUInt16(arrbFileData,    (nFileDataIndex += sizeof(System.Int16))),
                    dwSampleRate           = System.BitConverter.ToUInt32(arrbFileData,    (nFileDataIndex += sizeof(System.Int16))),
                    dwAverageBytesPerSecond = System.BitConverter.ToUInt32(arrbFileData,    (nFileDataIndex += sizeof(System.Int32))),
                    wBlockAlign            = System.BitConverter.ToUInt16(arrbFileData,    (nFileDataIndex += sizeof(System.Int32))),
                    wBitDepth              = System.BitConverter.ToUInt16(arrbFileData,    (nFileDataIndex += sizeof(System.Int16)))
                };

                // Find data chunk index
                if ((nFileDataIndex = Resources.IndexOf(arrbFileData,
                                                        Encoding.ASCII.GetBytes("data"),
                                                        (nFileDataIndex += sizeof(System.Int16)),
                                                        nFileDataIndex)) == -1)
                {
                    return (false);
                }

                // Fill in WAV data chunk information
                dcData = new WAV_DataChunk
                {
                    sGroupID    = Encoding.ASCII.GetChars(arrbFileData, nFileDataIndex, sizeof(byte) * Resources.GROUP_ID_LENGTH),
                    dwChunkSize = System.BitConverter.ToUInt32(arrbFileData,   (nFileDataIndex += sizeof(byte) * Resources.GROUP_ID_LENGTH)),
                };

                // Shift data index to sample data start
                nFileDataIndex += sizeof(System.Int32);

                // Decide upon the sample array type and load it with samples
                switch (fcFormat.wBitDepth / 8)
                {
                    case (sizeof(short)):
                        dcData.sampleData = Array.CreateInstance(typeof(short), (dcData.dwChunkSize * 8) / fcFormat.wBitDepth);
                        Samples.LoadSamples((short[])dcData.sampleData, arrbFileData, nFileDataIndex);
                        break;
                    case (sizeof(float)):
                        dcData.sampleData = Array.CreateInstance(typeof(float), (dcData.dwChunkSize * 8) / fcFormat.wBitDepth);
                        Samples.LoadSamples((float[])dcData.sampleData, arrbFileData, nFileDataIndex);
                        break;
                    default:
                        Unload();
                        return (false);
                }

                // Fill in basic file information
                sFilePath = sPathToFile;

                // Calculate new file size
                lFileSize = System.Runtime.InteropServices.Marshal.SizeOf(hHeader) +            // Header size
                            System.Runtime.InteropServices.Marshal.SizeOf(fcFormat) +           // Format chunk size
                            (sizeof(byte) * Resources.GROUP_ID_LENGTH) +                        // 'Group ID' variable type size
                            System.Runtime.InteropServices.Marshal.SizeOf(dcData.dwChunkSize) + // 'Chunk size' variable type size
                            dcData.dwChunkSize;                                                 // Actual sample data size in bytes

                // WAV File Duration = Data Chunk Size / Average Bytes per Second
                fFileDuration = (float)dcData.dwChunkSize / fcFormat.dwAverageBytesPerSecond;

                return (true);
            }

            return (false);
        }

        /// <summary>
        /// Unloads WAV instance.
        /// </summary>
        public void Unload()
        {
            // Reset class values
            hHeader     = new WAV_Header();
            fcFormat    = new WAV_FormatChunk();
            dcData      = new WAV_DataChunk();
            nFileBPM    = -1;
            sFilePath   = null;
        }

        /// <summary>
        /// Checks if a file is currently loaded within this instance.
        /// </summary>
        /// <returns>True if a file is loaded, False otherwise.</returns>
        public bool IsLoaded()
        {
            return (sFilePath != null);
        }

        /// <summary>
        /// Normalizes loaded audio.
        /// </summary>
        public void Normalize()
        {
            if (IsLoaded())
            {
```

```csharp
                // Normalize audio based on bit depth
                switch (fcFormat.wBitDepth / 8)
                {
                    case (sizeof(short)):
                        Samples.Normalize((short[])dcData.sampleData);
                        break;
                    case (sizeof(float)):
                        Samples.Normalize((float[])dcData.sampleData);
                        break;
                }
            }
        }

        /// <summary>
        /// Reverses polarity of loaded audio.
        /// </summary>
        public void ReversePolarity()
        {
            if (IsLoaded())
            {
                // Reverse polarity of audio based on bit depth
                switch (fcFormat.wBitDepth / 8)
                {
                    case (sizeof(short)):
                        Samples.ReversePolarity((short[])dcData.sampleData);
                        break;
                    case (sizeof(float)):
                        Samples.ReversePolarity((float[])dcData.sampleData);
                        break;
                }
            }
        }

        /// <summary>
        /// Reverses loaded audio.
        /// </summary>
        public void Reverse()
        {
            if (IsLoaded())
            {
                // Reverse audio based on bit depth
                switch (fcFormat.wBitDepth / 8)
                {
                    case (sizeof(short)):
                        Samples.Reverse((short[])dcData.sampleData);
                        break;
                    case (sizeof(float)):
                        Samples.Reverse((float[])dcData.sampleData);
                        break;
                }
            }
        }

        /// <summary>
        /// Writes loaded audio in its current state to given destination.
        /// </summary>
        /// <param name="sFileDestination">Path to write audio to (include extension).</param>
        /// <returns>True if file was written successfully, False otherwise.</returns>
        public bool Write(string sFileDestination)
```

```
            {
                // If file is currently loaded AND given file path is valid
                if (IsLoaded() &&
                    Resources.IsValidFilepath(sFileDestination))
                {
                    // Create a new output stream buffer
                    byte[] outStream = new byte[lFileSize];
                    int nOutStreamIndex;

                    // Write WAV header information
                    Encoding.ASCII.GetBytes(hHeader.sGroupID).CopyTo(outStream, (nOutStreamIndex = 0));
                    BitConverter.GetBytes(lFileSize - 8).CopyTo(outStream,          (nOutStreamIndex += sizeof(byte) * Resources.GROUP_ID_LENGTH));
                    Encoding.ASCII.GetBytes(hHeader.sRiffType).CopyTo(outStream, (nOutStreamIndex += sizeof(System.Int32)));

                    // Write WAV format chunk information
                    Encoding.ASCII.GetBytes(fcFormat.sGroupID).CopyTo(outStream, (nOutStreamIndex += sizeof(byte) * Resources.GROUP_ID_LENGTH));
                    BitConverter.GetBytes(fcFormat.dwChunkSize).CopyTo(outStream, (nOutStreamIndex += sizeof(byte) * Resources.GROUP_ID_LENGTH));
                    BitConverter.GetBytes(fcFormat.wFormatTag).CopyTo(outStream, (nOutStreamIndex += sizeof(System.Int32)));
                    BitConverter.GetBytes(fcFormat.wChannels).CopyTo(outStream, (nOutStreamIndex += sizeof(System.Int16)));
                    BitConverter.GetBytes(fcFormat.dwSampleRate).CopyTo(outStream, (nOutStreamIndex += sizeof(System.Int16)));
                    BitConverter.GetBytes(fcFormat.dwAverageBytesPerSecond).CopyTo(outStream,   (nOutStreamIndex += sizeof(System.Int32)));
                    BitConverter.GetBytes(fcFormat.wBlockAlign).CopyTo(outStream, (nOutStreamIndex += sizeof(System.Int32)));
                    BitConverter.GetBytes(fcFormat.wBitDepth).CopyTo(outStream, (nOutStreamIndex += sizeof(System.Int16)));

                    // Write WAV data chunk information
                    Encoding.ASCII.GetBytes(dcData.sGroupID).CopyTo(outStream,  (nOutStreamIndex += sizeof(System.Int16)));
                    BitConverter.GetBytes(dcData.dwChunkSize).CopyTo(outStream, (nOutStreamIndex += sizeof(byte) * Resources.GROUP_ID_LENGTH));

                    // Shift stream index to sample data start
                    nOutStreamIndex += sizeof(System.Int32);

                    // Write PCM samples to buffer
                    switch (fcFormat.wBitDepth / 8)
                    {
                        case (sizeof(short)):
                            Samples.WriteSamples(outStream, (short[])dcData.sampleData, nOutStreamIndex);
                            break;
                        case (sizeof(float)):
                            Samples.WriteSamples(outStream, (float[])dcData.sampleData, nOutStreamIndex);
                            break;
                    }

                    // Write byte stream to disk
                    File.WriteAllBytes(sFileDestination, outStream);

                    // Return true if file was written successfully
                    return (File.Exists(sFileDestination));
                }

                return (false);
            }

            /// <summary>
            /// Returns information about the currently loaded audio.
            /// </summary>
            /// <returns>Returns information about the currently loaded audio.</returns>
            public override string ToString()
            {
                if (IsLoaded())
                {
                    return ("Path: " + sFilePath + "\n" +
                            "Duration: " + fFileDuration + " (s)\n" +
                            "Size: " + lFileSize / (float)1000 + " (kb)\n" +
                            "BPM: " + BPM + "\n" +
                            "Sample Rate: " + fcFormat.dwSampleRate + " (Hz)\n" +
                            "Bit Depth: " + fcFormat.wBitDepth + " (bits per sample)\n" +
                            "Channels: " + fcFormat.wChannels);
                }

                return ("No file loaded.");
            }
        }
    }
```

2. מפרטי חומרה

<u>ASUS ZenBook UX305</u> – מחשב

- CPU: 900MHz Intel Core M3-6Y30 (dual-core, 4MB cache, 2.2GHz with Turbo Boost)
- Graphics: Intel HD Graphics 515
- RAM: 8GB DDR3L (1,866 MHz SDRAM)
- Screen: 13.3-inch, 1,920 x 1,080 IPS display
- Storage: 256GB SSD (M.2 2280)
- Ports: 3 x USB 3.0, SD card reader, micro HDMI, headset jack
- Connectivity: Integrated 802.11ac
- Camera: 2MP HD webcam
- Weight: 2.6 pounds (1.18kg)
- Size: 12.8 x 8.9 x 0.48 inches (32.5 x 22.6 x 1.23cm; W x D x H)

<u>Samsung SM-A7000</u> – מכשיר אנדרואיד

- CPU: Octa-core (4x1.5 GHz Cortex-A53 & 4x1.0 GHz Cortex-A53)
- Graphics: Adreno 405
- OS: Android 5.0 (Lollipop)
- RAM: 2GB

<u>Arduino Nano</u> – בקר ארדוינו

- Microcontroller: ATmega328
- Operating Voltage: 5V
- Input Voltage: 5-12V
- Digital I/O Pins: 14
- Analog Input Pins: 8
- Flash Memory: 32KB
- Clock Speed: 16MHz

3. מילון מונחים

DSP (Digital Signal Processing) - תחום בהנדסת חשמל, אלקטרוניקה ופיזיקה העוסק באותות, בייצוגם הספרתי ובשיטות העיבוד של אותות אלה.

Fourier Transform - כלי מרכזי באנליזה הרמונית שאפשר לתארו כפירוק של פונקציה לרכיבים מחזוריים (סינוסים וקוסינוסים או לחלופין אקספוננטים מרוכבים) וביצוע אנליזה מתמטית לפונקציה על ידי ניתוח רכיביה.

Sampling - עיבוד של אות רציף לכדי אות בדיד. דוגמה נפוצה לדגימה היא המרה של גלי קול (אות רציף בזמן) לרצף של דגימות בדידות בזמן. דגימה מייחסת ערך או קבוצת ערכים בדידים למשתנה התלוי (לרוב זמן או מרחב).

BPM - מונח מוזיקלי המגדיר את המהירות/קצב של לחן מסוים, וכן מונח המגדיר קצב פעימות הלב.

Signal - תנודה, גודל כלשהו התלוי בזמן או במרחב, או הפרעה, בעלי משמעות כלשהי.

4. מקורות

RIFF & WAVE Format

https://blogs.msdn.microsoft.com/dawate/2009/06/23/intro-to-audio-programming-part-2-demystifying-the-wav-format/
http://soundfile.sapp.org/doc/WaveFormat/
https://en.wikipedia.org/wiki/WAV
https://en.wikipedia.org/wiki/Resource_Interchange_File_Format

Fourier Transform

https://en.wikipedia.org/wiki/Fast_Fourier_transform
https://en.wikipedia.org/wiki/Discrete_Fourier_transform
http://www.dspguide.com/ch12/2.htm
https://arxiv.org/html/math/0302212
http://www.dspguide.com/ch12/2.htm

Hardware

https://pulsesensor.com/
https://github.com/WorldFamousElectronics
https://pulsesensor.com/pages/code-and-guide
http://www.instructables.com/id/Arduino-AND-Bluetooth-HC-05-Connecting-easily/

Tools

https://www.draw.io
http://www.planetb.ca/syntax-highlight-word