TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA
ROMANIA

Faculty of Automation and Computers

Specialization: Computer Science

Arduino music

Piano and music player with Arduino Mega

The following project presents the design steps and functioning of a piano with an option for playing hardcoded songs. This is an original approach.

Teaching Assistant: Rednic Ana

Author: Coroian Tudor

Arduino music

Piano and music player with Arduino Mega

1. Introduction

In today's world, music is all around us. Be it a jingle for a TV commercial, a ring tone for the phone that is always in our pockets or a silly toy bought from a thrift shop, songs are always present. It was only natural for me to "transpose"[1] this behavior into a project which has nothing to do with music.

1.1 Why did I choose this project?

I have always been fascinated by music and sounds in general. The way in which different frequencies interact to produce various auditory sensations is a topic on which I still have a lot to learn.

In other words, I took it as a challenge. I had to use components which were designed mainly for debugging or state/error indicators (i.e. buzzers) and use them to create something as closely related to music as possible. The task proved successful and has plenty of room for adding more songs, in as almost-user-friendly manner.

1.2 Other existing solutions

To my knowledge, my solution is unique. Although it appears basic at first (meaning there aren't many ways for the main part of the project to be implemented), the approach I took was solely based on what I knew was possible on the Arduino Mega board and with the components that I had available.

This being said, I do not claim that "nobody implemented this". Building a piano from an Arduino board seems like a common idea. As such, I like to think of my approach as being "on the same note, but from another octave[2]".

Contents

[1] Word play; "to transpose" (music.) means to shift all notes from an interval with the same number of semitones

[2] Word play; two notes can have the same name, but different frequencies, thus being from different octaves

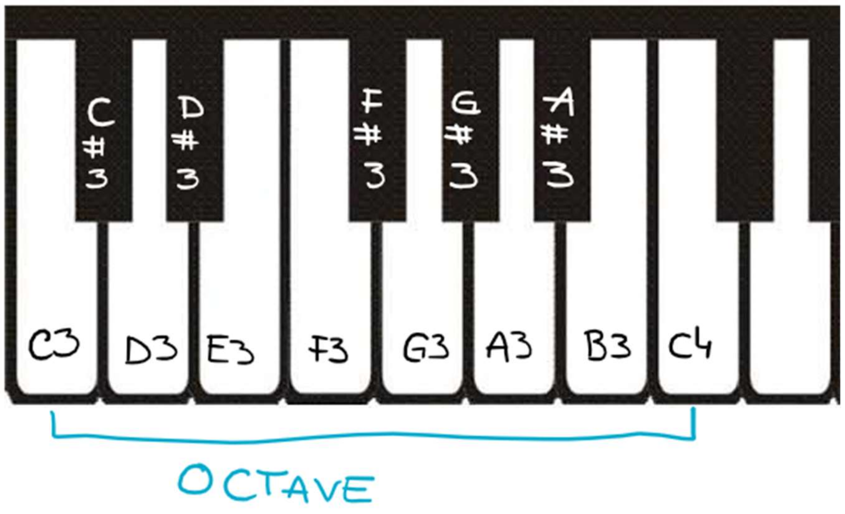
2. Bibliographic research

In order to have the reader understand the implementation of the project in-depth, a short background in music theory is needed.

2.1 Theoretical background on music

Not to worry the reader, this is not a full course on music theory, just a series of definitions of the most important notions (i.e. only the ones used in this project).

2.1.1 Intervals



The distance between any two notes of the piano (e.g. between a white one and a black one, or between two white ones that have no black notes between them) is called a **semitone** (or half step). For the purpose of this project, we consider the semitone as the smallest possible interval between two notes.

An **octave** is the distance between two notes with the same name, but different numbers. Alternatively, one can think of the octave as being the interval formed by 12 semitones.

The "#" (sharp) notation is used to mark a note which is a semitone higher than the note represented by the letter that it is attached to.

The " ♭ " (flat) notation is used to mark a note which is a semitone lower than then note represented by the letter that it is attached to.

For the sake of simplicity, I will use only the sharp notation throughout this document.

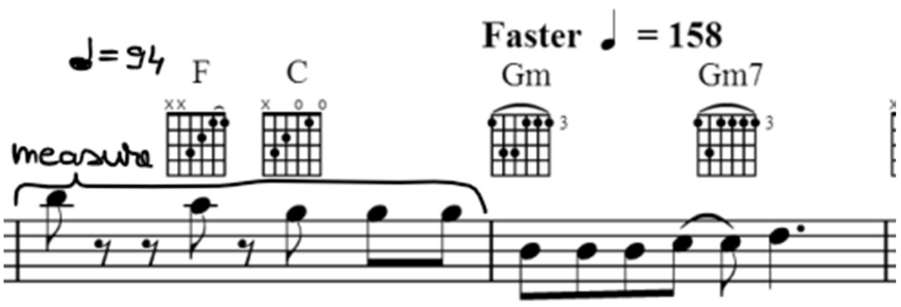
Contents

2. Bibliographic research

2.1.2 Note duration

In music theory, the duration of notes is not absolute, but rather relative. A quarter note can have two different durations in two different songs. As such, each song has a tempo specified at the beginning of the piano sheet.

The **tempo** is a measurement unit for the speed at which a passage of music should be played. To be noted that a single song can have multiple tempos. In this case, the new tempo is specified on top of the measure from which the new tempo is applied.



The tempo marks the number of quarter notes played in one minute. Therefore, a tempo like ♩ = 120 should be read as "120 quarter notes per minute".

I have mentioned that in music, notes have relative durations. All of them are based on the duration of the quarter note. As such, here are the most used notes:

Name	Note	Rest
Whole Note	𝅝	𝄻
Half Note	𝅗𝅥	𝄼
Quarter Note	♩	𝄽
Eighth Note	♪	𝄾
Sixteenth Note	𝅘𝅥𝅯	𝄿

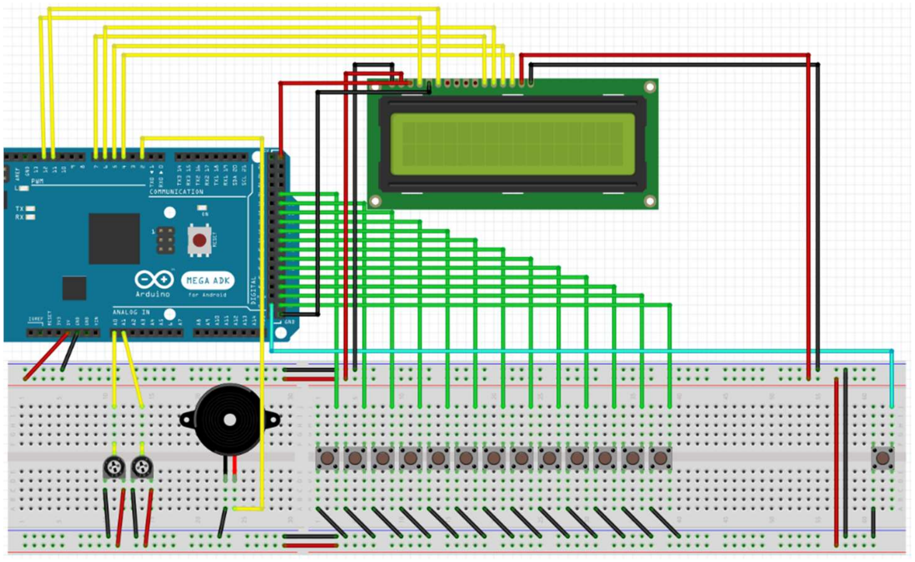
A whole note is 4x the quarter note, a half note is 2x the quarter note, an eight note is 1/2x the quarter note and so on. Each note has a corresponding **rest** duration or pause (no note is played).

Contents

3. Proposed solution

With the theoretical stuff out of the way, the focus now becomes the project itself.

3.1 Overview



In the picture above, the final stage of the project is presented. The components used are as follows:

- Arduino Mega board
- piezoelectric buzzer
- 2 x potentiometer
- 14 x button
- LCD
- wires
- 2 x breadboard

Starting with the buttons, the first noticeable aspect about them is that they come into two groups. The thirteen buttons on the left correspond to an entire octave on the piano. The button on the right is used for playing one of the loaded songs.

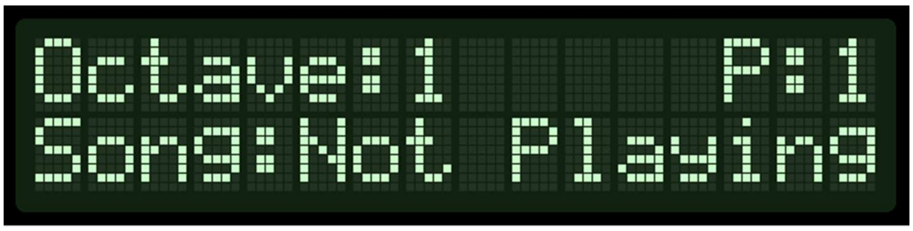
The piezoelectric buzzer is the one that emits the sounds corresponding to each button.

The potentiometer on the left is used to change the octaves (from 1 to 7) and the potentiometer on the right is used to cycle between the loaded songs.

Contents

3. Proposed solution

The LCD is used for display purposes. It will be analyzed next:



The number after the octave refers to the number of the octave that is currently selected from the potentiometer. The number after "P" refers to the number of the song that will be played when the button is pressed and the "Not Playing" status will be updated to display the number of the song.

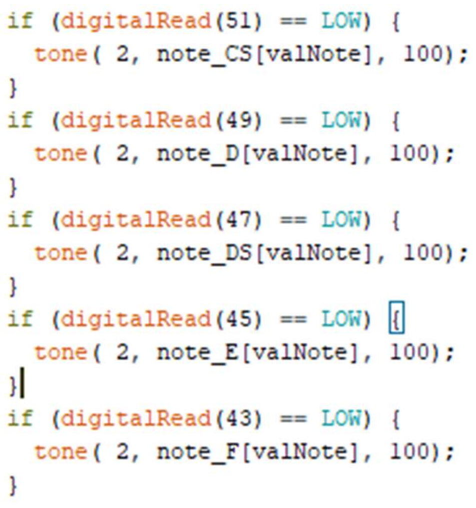
3.2 Algorithms and implementation

In what regards the algorithms used for this project, they should be grouped in two categories:

- piano related algorithms
- playback related algorithms

3.2.1 Piano

For the piano algorithm I went for 13 "if" structures, one for each button. This allows the program to play each note individually.



```
if (digitalRead(51) == LOW) {
  tone( 2, note_CS[valNote], 100);
}
if (digitalRead(49) == LOW) {
  tone( 2, note_D[valNote], 100);
}
if (digitalRead(47) == LOW) {
  tone( 2, note_DS[valNote], 100);
}
if (digitalRead(45) == LOW) {
  tone( 2, note_E[valNote], 100);
}
if (digitalRead(43) == LOW) {
  tone( 2, note_F[valNote], 100);
}
```

The snippet above present exactly what was described before. Based on the "valNote" variable, the program takes from a vector the corresponding note. For the last note in the octave (i.e. the note which is exactly one octave above the first note) the program uses the value "valNote + 1".

Contents

3. Proposed solution

The vectors which store the notes are coded as follows:

```
int note_C[8] = {NOTE_C1, NOTE_C2, NOTE_C3, NOTE_C4, NOTE_C5, NOTE_C6, NOTE_C7, NOTE_C8};
int note_CS[7] = {NOTE_C1, NOTE_CS2, NOTE_CS3, NOTE_CS4, NOTE_CS5, NOTE_CS6, NOTE_CS7};
int note_D[7] = {NOTE_D1, NOTE_D2, NOTE_D3, NOTE_D4, NOTE_D5, NOTE_D6, NOTE_D7};
int note_DS[7] = {NOTE_DS1, NOTE_DS2, NOTE_DS3, NOTE_DS4, NOTE_DS5, NOTE_DS6, NOTE_DS7};
int note_E[7] = {NOTE_E1, NOTE_E2, NOTE_E3, NOTE_E4, NOTE_E5, NOTE_E6, NOTE_E7};
int note_F[7] = {NOTE_F1, NOTE_F2, NOTE_F3, NOTE_F4, NOTE_F5, NOTE_F6, NOTE_F7};
int note_FS[7] = {NOTE_FS1, NOTE_FS2, NOTE_FS3, NOTE_FS4, NOTE_FS5, NOTE_FS6, NOTE_FS7};
int note_G[7] = {NOTE_G1, NOTE_G2, NOTE_G3, NOTE_G4, NOTE_G5, NOTE_G6, NOTE_G7};
int note_GS[7] = {NOTE_GS1, NOTE_GS2, NOTE_GS3, NOTE_GS4, NOTE_GS5, NOTE_GS6, NOTE_GS7};
int note_A[7] = {NOTE_A1, NOTE_A2, NOTE_A3, NOTE_A4, NOTE_A5, NOTE_A6, NOTE_A7};
int note_AS[7] = {NOTE_AS1, NOTE_AS2, NOTE_AS3, NOTE_AS4, NOTE_AS5, NOTE_AS6, NOTE_AS7};
int note_B[7] = {NOTE_B1, NOTE_B2, NOTE_B3, NOTE_B4, NOTE_B5, NOTE_B6, NOTE_B7};
```

The values are defined at the beginning of the document and they represent the corresponding frequencies of each note. The “S” added to the 7 base notes represents the sharp symbol that was described previously.

The “valNote” variable is mapped from the input of the potentiometer, as follows:

```
int valNote = map(val , 0, 1000, 0, 6);
```

3.2.2 Playback

The playback system works based on the same notes that were presented before. Each song is hardcoded into a matrix, as follows:

```
int melodies[][45] = {
  {NOTE_D4, NOTE_D4, NOTE_D4, NOTE_D4, NOTE_D4, NOTE_G4,
   NOTE_F4, NOTE_D4, NOTE_G4, NOTE_F4, NOTE_F4, NOTE_C4,
   NOTE_D4, NOTE_D4, NOTE_D4, NOTE_D4, NOTE_D4, NOTE_G4,
   NOTE_F4, NOTE_D4, NOTE_G4, NOTE_F4, NOTE_C4, NOTE_D4},
```

The number of songs is updated dynamically, based on the number of songs written in the “melodies” matrix.

As it was described in the previous chapter, each song has its own tempo. In this context, I implemented a “tempos” vector which holds the tempo for each individual song.

```
int tempos [5] = {120, 120, 94, 158, 175};
```

In case a song has multiple tempos within it, the program will address this inside the playing algorithm itself.

Since each note duration is relative in music, I implemented a matrix similar to the “melodies” one, but this time storing the relative duration of each note. As such, “1” represents a quarter note and the other numbers are all multiples or fractions of this.

Contents

3. Proposed solution

```
float durations[][45] = {
  {1, 1, 1, 0.5, 0.5, 0.75, 0.75,
   0.25, 0.75, 0.25, 0.5, 0.5,
   1, 1, 1, 0.5, 0.5, 0.75, 0.75,
   0.25, 0.75, 0.75, 0.5, 1},
```

As it can be seen from the snippet, some number are not powers of two. This is because in music, notes can appear with a small point after them. This point means that to the duration of the current note, we add half of it. In this manner, an eighth note with a point after it will be translated to 0.5 + ½ * 0.5 = 0.75.

However, simply writing the relative duration of the notes doesn't help with much. The "tone()" command expects the user to input the exact time (in microsecond). This is where the tempo of the song comes in. Having all the tempos stored in one vector and all the durations stored in one matrix, we can easily access them based on a common index.

For the transformation of the relative duration into microseconds, I wrote the formula:

$$\frac{60\ 000}{tempo} \times duration$$

This formula is then applied inside the algorithm that plays the song, as follows:

```
tone( 2, melodies[valSong][i], (60000/(tempos[valSong])) * durations[valSong][i]);
delay((60000/(tempos[valSong])) * durations[valSong][i]);
```

In the next snippet, the way in which the program deals with songs having multiple tempos is presented:

```
if (valSong == 2 && i < 17) {
  float duration = 60000 / tempos[valSong] * durations[valSong][i];
  tone( 2, melodies[valSong][i], duration);
  Serial.print(duration);
  Serial.print("\n");
  delay(duration);
} else if (valSong == 2 && i > 17) {
  float duration = 60000 / 158 * durations[valSong][i];
  tone( 2, melodies[valSong][i], duration);
  delay(duration);
```

The third song has two tempos. The first one end on the 17th note and the second one begins on the same note. This problem was solved with a single if structure.

Contents
• • •

3. Proposed solution

3.3 Details

In order for each note of the song to be distinguishable from the previous one, I added a small delay between notes that is equal to the duration of the previous note. Through testing, I found out that this was the best approach.

I also tried making the pause between note be equal to the remaining value until the next note duration (e.g. 0.75 will have a delay of 0.25). However, this failed on the notes that were already integer themselves (e.g. the quarter note).

All the code presented above is either a declaration found outside of all the methods or an actual algorithm found inside the "loop()" function.

Contents

4. Testing

The testing of the project was conducted in three big steps:

- first run
- playback system
- display

4.1 First run

In the first run of the project, I only connected the buttons and attached to them one note. This was done in order to test if the approach with multiple "if" structures is suitable for this project.

The test proved to be successful, since it's the algorithm that is currently found in the latest version on the project.

4.2 Improvement steps

After running the code as presented before, I added the first potentiometer and wrote the vectors that holds the notes that are one octave apart. Another test was run after this update, which at first failed, because of the way the values were mapped from the potentiometer.

I tweaked the "map()" function to only map the interval between 0 to 1000, instead of 1023, which proved to be the best way to approach the situation.

After this, I started to design and implement the playback system. At first, I thought of writing each song as a separate vector, but this became a problem, because a lot of code was repeating. After the third song, I switched the approach and I wrote every song inside the matrix, in the form that it is now.

At each of these steps, I ran the code on the Arduino Mega board to make sure that everything was working properly.

A lot of time was spent at this step, because it was a tedious work. If it happened for me to miss a note or a duration, the test would not fail, but it would have not sounded as intended.

At last, after everything was working properly, I began connecting and implementing the last part of the project, the display.

Contents

4. Testing

Testing, at this point became easier, since it basically worked from the first try and any more tests that were run, were just to ensure a good aesthetic of the output on the display.

4.3 Final state

The code was run one last time in the final state to make sure that everything works properly.

Each of the songs were played and each of the notes as well. This was done in order to ensure that there is no fault in any of the running scenarios of the project.

Also worth mentioning at this point is that the layout of the components on the breadboard was made in such a way to allow for easy access to all the buttons and the potentiometers.

Contents

5. Conclusions

The purpose of the project was to have a working piano. By the time I was done with the project, I had much more than that.

This project has an entertainment purpose, since it can be seen as a toy. The sounds made by the piezoelectric buzzer resemble the sounds made by most of the small toys on the market.

The greatest achievement of the project is that it can be easily expandable from the code itself. If the user wants to add more songs to the list, the procedure is simple:

- translate the notes from the piano sheet into the melody matrix (write 0 for pauses within the notes)
- write the relative durations inside the duration matrix (positional notation)
- write the tempo in the tempos vector, on the position corresponding to the song currently written

Overall, this was a great project to construct and implement.

Contents