WPF Midi Band

codeproject.com/Articles/141617/WPF-Midi-Band

Marcelo Ricardo de Oliveira, 3 Jan 2011

★★★★★ 4.98 (64 votes)

Learn how to play midi files using fun WPF animations

Go to top

- Download source - 741.62 KB
- Download EXE - 635.37 KB

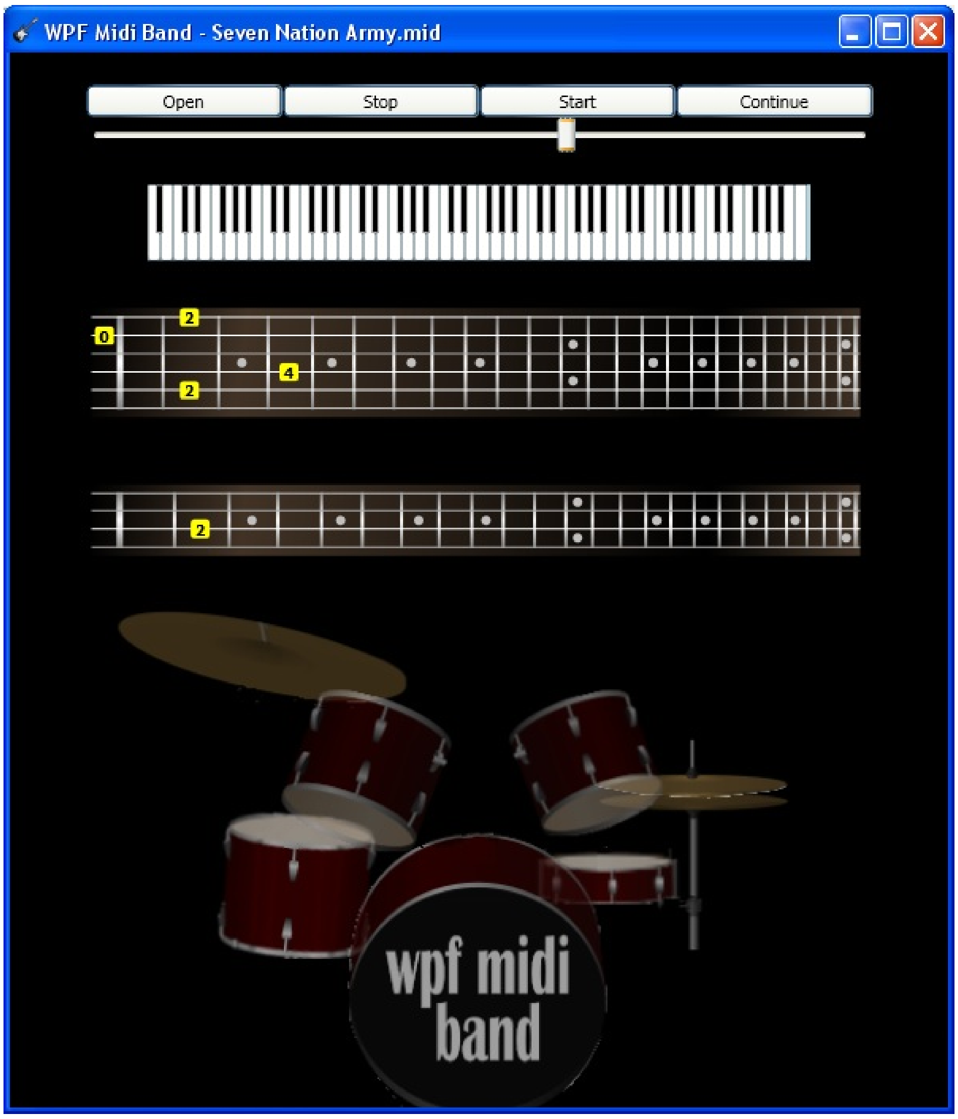


Table of Contents

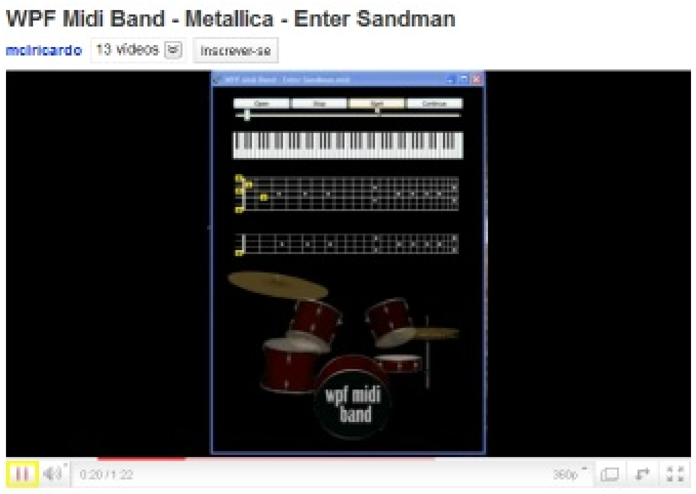
Introduction

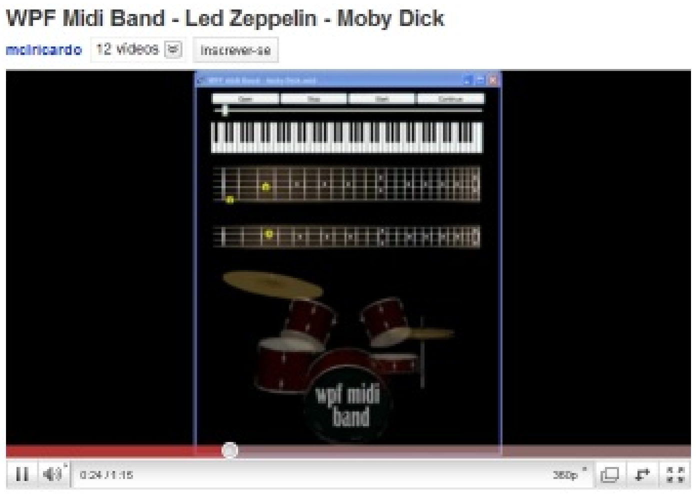
Since a few months ago, when I started studying WPF, I have noticed that there aren't many articles dealing with WPF and audio features. This is more noticeable when it comes to Midi audio. So, scarce material on the issue became the main motivation for this article. The secondary motivation, of course, is that it is a lot of fun to work with Midi, as I found out later.

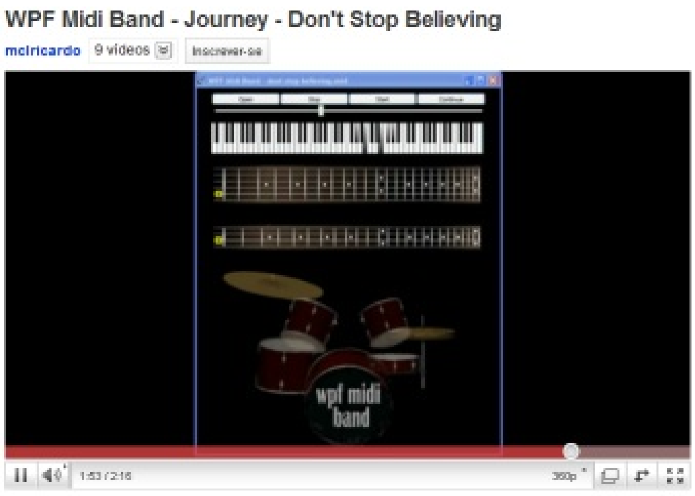
Although the following article is not written by an expert on music and audio engineering, I hope to provide developers and users in general at least with the basic concepts involving C# and midi.

Youtube Videos

To find out quickly how the application works, you can take a look at the Youtube video samples below:






WPF Midi Band - Journey - Don't Stop Believing

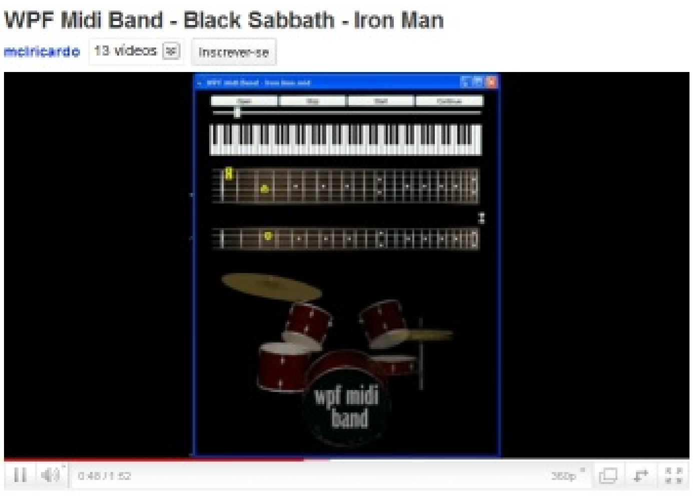
mclricardo 9 videos Inscrever-se

WPF Midi Band - Lady Gaga - Poker Face

mclricardo 11 videos Inscrever-se

WPF Midi Band - Queen & David Bowie - Under Pressure

mclricardo 12 videos Inscrever-se



System Requirements

To use WPF Midi Band provided with this article, if you already have Visual Studio 2010, that's enough to run the application. If you don't, you can download the following 100% free development tool directly from Microsoft:

Visual C# 2010 Express

Why Midi?

At this point, you might be wondering why I decided to workMidi. Wouldn't be better to use more high quality music provided by MP3?

Well, to answer that, I must say that, as an IT professional, what made Midi files appealing for me is that they contain lots of data. Quality data. And these data may be processed in many ways, as you can see in this article. That being said, although Midi music often lacks the beauty and depth of traditional instruments, it is in many ways like a programming language (like C#) where we have commands, arguments and enumerations. In short, it is a program that plays music.

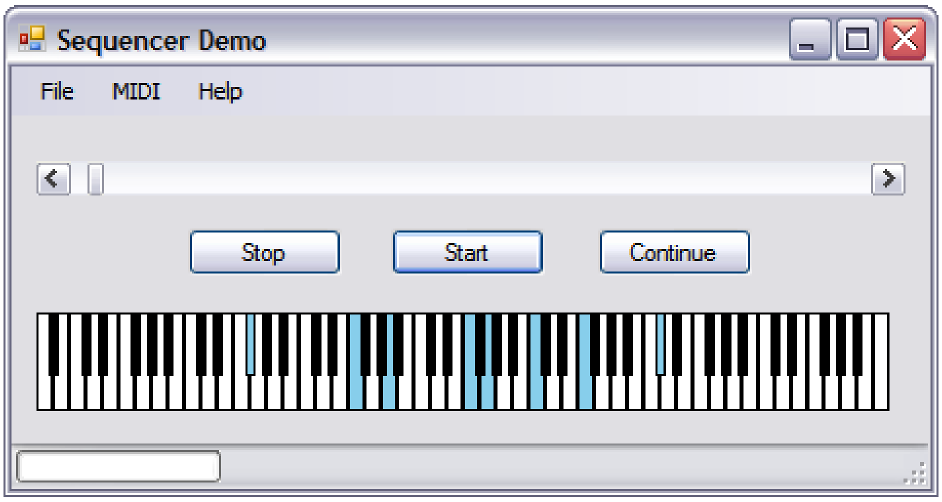
Midi Overview

Midi stands for **Musical Instrument Digital Interface**. It is an electronic protocol established in 1982 aiming to standardize the growing electronic instruments industry of the time, and adopted widespread until these days. Midi protocol is not about audio signal. Instead, it is based on event messages, and sends commands to electronic devices, such as which instruments to play, which channel, volume, pitch, define tempo, and so on.

I must admit that, before working in this article, I didn't have much love for Midi music. This is because some musics have an instrument, pitch or volume that can be annoying. But there are also many quality midis out there. It's up to you to find them.

The C# Midi Toolkit

This article would not be possible without the priceless contribution of an article published here in The Code Project years ago by **Leslie Sanford**. Leslie wrote this article entitled C# Midi Toolkit and it became a must read for C# developers wanting to get more out of Midi.



Leslie Sanford's C# Midi Toolkit

The core of WPF Midi Band application uses Leslie's excellent Toolkit, so if you are really interested in the subject, I strongly recommend reading C# Midi Toolkit.

Basically, you have to follow the steps below to playmidi with C# Midi Toolkit:

- Instantiate a `OutputDevice`
- Instantiate a `Sequence`
- Instantiate a `Sequencer`
- Subscribe the `Sequencer.ChannelMessagePlayed` event
- In the `Sequencer.ChannelMessagePlayed` event, call the `outDevice.Send`, passing the message to the device as an argument
- Attach the `Sequence` to the `Sequencer`
- Instantiate an `OutputDevice`
- Call `Sequencer.LoadAsync` passing a file name as argument
- Listen to the music

There are other methods, of course, but these steps are basic ones. The full implementation is as follows:

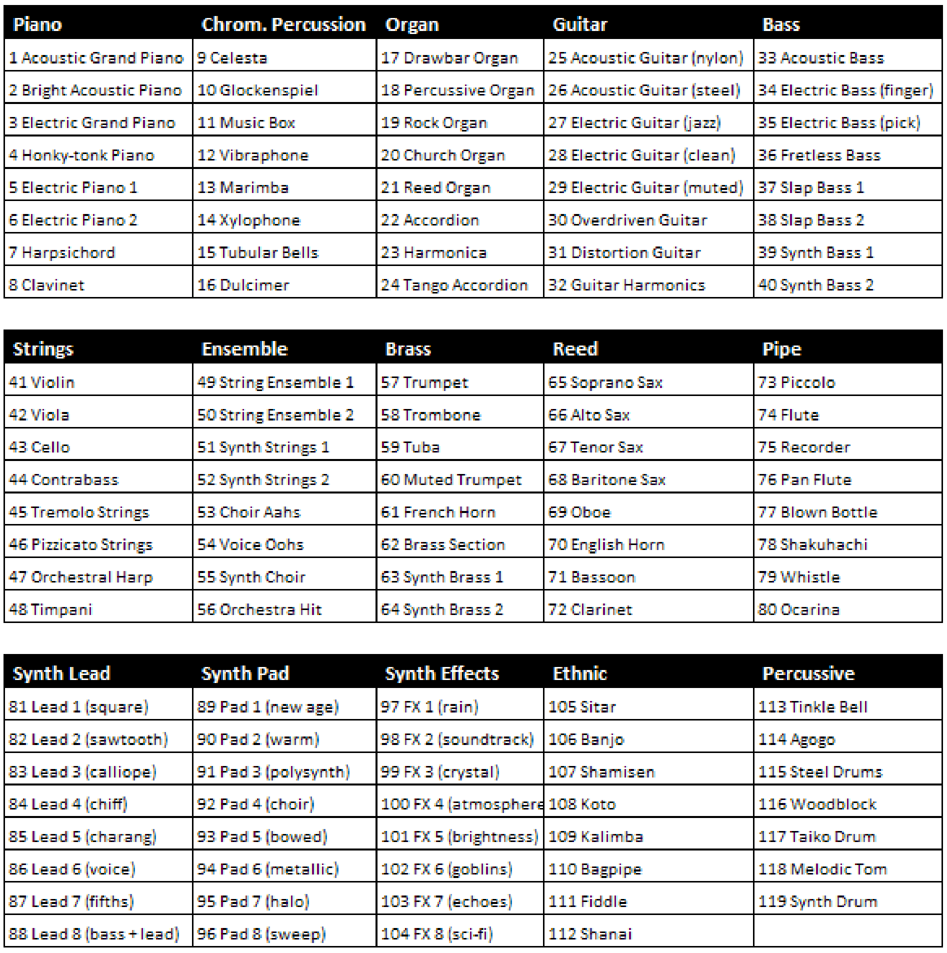
Hide Shrink ▲ Copy Code

```
public partial class MainWindow : Window
{
 ...
 private OutputDevice outDevice;
 private Sequence sequence1 = new Sequence();
 private Sequencer sequencer1 = new Sequencer();

 public MainWindow()
 {
  ...
  InitializeSequencer();
  ...
 }

 private void InitializeSequencer()
 {
  ...
  outDevice = new OutputDevice(outDeviceID);
  this.sequence1.Format = 1;
  this.sequencer1.Position = 0;
  this.sequencer1.Sequence = this.sequence1;
  this.sequencer1.PlayingCompleted +=
  new System.EventHandler(this.HandlePlayingCompleted);
  this.sequencer1.ChannelMessagePlayed +=
  new System.EventHandler<sanford.multimedia.midi.channelmessageeventargs>
  (this.HandleChannelMessagePlayed);
  this.sequencer1.Stopped +=
  new System.EventHandler<sanford.multimedia.midi.stoppedeventargs>
  (this.HandleStopped);
  this.sequencer1.SysExMessagePlayed +=
  new System.EventHandler<sanford.multimedia.midi.sysexmessageeventargs>
  (this.HandleSysExMessagePlayed);
  this.sequencer1.Chased +=
  new System.EventHandler<sanford.multimedia.midi.chasedeventargs>
  (this.HandleChased);
  this.sequence1.LoadProgressChanged += HandleLoadProgressChanged;
  this.sequence1.LoadCompleted += HandleLoadCompleted;
 }

 private void HandleChannelMessagePlayed(object sender,
ChannelMessageEventArgs e)
 {
  ...
  outDevice.Send(e.Message);
  ...
 }
}
```

Instruments

This article will make use ofMidi event messages to show how to play the Midi music as a virtual band called**WPF Midi Band**. Please notice that we are dealing with the Midi data *reactively*, so instead of opening the file and reading the data directly, we instead tell the `Sequencer` to open the file asynchronously, and then all we have to do is to wait and listen to `Sequencer` events. Whenever the `Sequencer` that a `ChannelMessage` has been played, we immediately send that message to the `OutputDevice` , which will cause a sound to be played, or stopped, or distorted, and so on.

Below we have the table containing the "Melodic Sounds" supported byMidi protocol:



Piano	Chrom. Percussion	Organ	Guitar	Bass
1 Acoustic Grand Piano	9 Celesta	17 Drawbar Organ	25 Acoustic Guitar (nylon)	33 Acoustic Bass
2 Bright Acoustic Piano	10 Glockenspiel	18 Percussive Organ	26 Acoustic Guitar (steel)	34 Electric Bass (finger)
3 Electric Grand Piano	11 Music Box	19 Rock Organ	27 Electric Guitar (jazz)	35 Electric Bass (pick)
4 Honky-tonk Piano	12 Vibraphone	20 Church Organ	28 Electric Guitar (clean)	36 Fretless Bass
5 Electric Piano 1	13 Marimba	21 Reed Organ	29 Electric Guitar (muted)	37 Slap Bass 1
6 Electric Piano 2	14 Xylophone	22 Accordion	30 Overdriven Guitar	38 Slap Bass 2
7 Harpsichord	15 Tubular Bells	23 Harmonica	31 Distortion Guitar	39 Synth Bass 1
8 Clavinet	16 Dulcimer	24 Tango Accordion	32 Guitar Harmonics	40 Synth Bass 2

Strings	Ensemble	Brass	Reed	Pipe
41 Violin	49 String Ensemble 1	57 Trumpet	65 Soprano Sax	73 Piccolo
42 Viola	50 String Ensemble 2	58 Trombone	66 Alto Sax	74 Flute
43 Cello	51 Synth Strings 1	59 Tuba	67 Tenor Sax	75 Recorder
44 Contrabass	52 Synth Strings 2	60 Muted Trumpet	68 Baritone Sax	76 Pan Flute
45 Tremolo Strings	53 Choir Aahs	61 French Horn	69 Oboe	77 Blown Bottle
46 Pizzicato Strings	54 Voice Oohs	62 Brass Section	70 English Horn	78 Shakuhachi
47 Orchestral Harp	55 Synth Choir	63 Synth Brass 1	71 Bassoon	79 Whistle
48 Timpani	56 Orchestra Hit	64 Synth Brass 2	72 Clarinet	80 Ocarina

Synth Lead	Synth Pad	Synth Effects	Ethnic	Percussive
81 Lead 1 (square)	89 Pad 1 (new age)	97 FX 1 (rain)	105 Sitar	113 Tinkle Bell
82 Lead 2 (sawtooth)	90 Pad 2 (warm)	98 FX 2 (soundtrack)	106 Banjo	114 Agogo
83 Lead 3 (calliope)	91 Pad 3 (polysynth)	99 FX 3 (crystal)	107 Shamisen	115 Steel Drums
84 Lead 4 (chiff)	92 Pad 4 (choir)	100 FX 4 (atmosphere	108 Koto	116 Woodblock
85 Lead 5 (charang)	93 Pad 5 (bowed)	101 FX 5 (brightness)	109 Kalimba	117 Taiko Drum
86 Lead 6 (voice)	94 Pad 6 (metallic)	102 FX 6 (goblins)	110 Bagpipe	118 Melodic Tom
87 Lead 7 (fifths)	95 Pad 7 (halo)	103 FX 7 (echoes)	111 Fiddle	119 Synth Drum
88 Lead 8 (bass + lead)	96 Pad 8 (sweep)	104 FX 8 (sci-fi)	112 Shanai	

Midi Melodic Sounds Table

The instrument selection done by sending a `ProgramChange` command in the event message to the output device. When you send `ProgramChange` to the device, you define also the channel number reserved for that particular instrument.

For example, the snippet below is an XML serialization of a single message, taken from the beginning of "Sweet Child of Mine" Midi:

Hide Copy Code

```
<Message>
  <id>10</id>
  <segmentId>0</segmentId>
  <ChannelCommand>ProgramChange</ChannelCommand>
  <MidiChannel>3</MidiChannel>
  <Data1>35</Data1>
  <Data2>0</Data2>
  <MessageType>Channel</MessageType>
</Message>
```

Notice that the `ProgramChange` command selects the instrument with Id 35 (Data1 = 35, so the instrument is Electric Bass - pick) in the Channel #3 (MidiChannel = 3). This single message prepares the output device to process any incoming notes events arriving at the Channel 3 as Electric Bass notes.

Instrument Selection

As the messages sent to the output device trigger the `HandleChannelMessagePlayed` back in our code behind class, we have to decide which instrument (in the screen) should receive those messages.

The first thing we have to check is whether the `ChannelCommand` is a `ProgramChange` command. In this case, we have to feed a dictionary to store the Instrument Id, so that the future messages coming to that particular channel could be directed to the correct instruments:

Hide   Copy Code

```
private void HandleChannelMessagePlayed(object sender,
ChannelMessageEventArgs e)
{
 ...
 if (e.Message.Command == ChannelCommand.ProgramChange)
 {
  if (!dicChannel.ContainsKey(e.Message.MidiChannel))
  {
   dicChannel.Add(e.Message.MidiChannel, e.Message.Data1);
  }
 }
 ...
```

The second thing to be taken into consideration is whether a drum sound is being played or not. This is done by inspecting the `MidiChannel` and checking if the message is coming to the 10th Channel. Since MidiChannel is a 0-based data, we look for MidiChannel=9. This particular channel is reserved for drum sounds:

Hide   Copy Code

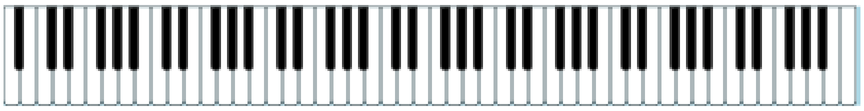
```
private void HandleChannelMessagePlayed(object sender,
ChannelMessageEventArgs e)
{
 ...
 if (e.Message.MidiChannel == 9)
 {
  this.Dispatcher.Invoke(
    System.Windows.Threading.DispatcherPriority.Normal,
   new Action(
    delegate()
    {
     drumControl1.Send(e.Message);
    }
   ));
 }
 else
 {
```

Notice above how we send the incoming message to the `DrumsControl`. Notice also that we have to deal with the execution coming from another thread different from the UI thread.

If the incoming message is not a drum sound, we should check the `MidiChannel` number. Then, based on the Melodic Sounds table shown earlier in this article, we decide which control (in the screen) should receive that message.

The Keyboard Control

The Piano Control resembles the **C# Midi Toolkit** original piano. The result is almost the same. The main difference, is that I ported it from Windows Forms to a WPF interface. The Piano Control will play any keyboard-like instrument, such as Grand Piano, Keyboard, Clavinet, etc. It also plays woodwind and tubular instruments such as flute, brass and saxophone. It could be a good idea that WPF Midi Band had one control for each instrument, but this would certainly cause visual confusion. Besides, in most classic rock bands (like WPF Midi Band), such instruments would probably play in an electronic keyboard.


WPF Midi Band's Keyboard

When selecting the instrument, we send the incoming message to the `PianoControl` based on the following conditions:

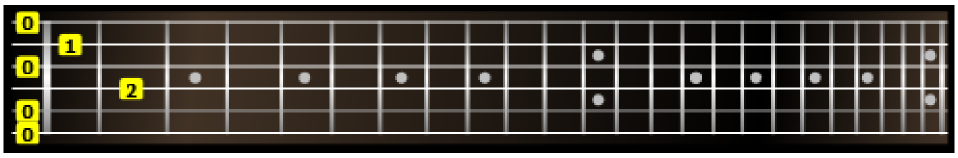
Hide   Copy Code

```
switch (dicChannel[e.Message.MidiChannel])
{
 case (int)MIDIInstrument.AcousticGrandPiano:
 case (int)MIDIInstrument.BrightAcousticPiano:
 case (int)MIDIInstrument.ElectricGrandPiano:
 case (int)MIDIInstrument.HonkytonkPiano:
 case (int)MIDIInstrument.ElectricPiano1:
 case (int)MIDIInstrument.ElectricPiano2:
 case (int)MIDIInstrument.Harpsichord:
 case (int)MIDIInstrument.Clavinet:
  pianoControl1.Send(e.Message);
   break;
```

The Guitar Control

The Guitar Control is a `UserControl` made of a guitar arm, containing 6 strings and a series of frets. It has the traditional guitar tuning (EADGBE), but you could replace it by another tuning if you like. This is not configurable by default, so you should put your hands on the code and recompile the application.

Each of the strings contain a sequence of notes. Each note falls between 2 frets and in a specific string. The application calculates which interval the note belongs to and then decides where the visual element representing the note will be shown.



WPF Midi Band's Guitar

The messages are sent to the `GuitarControl` only if the instruments are compatible with the guitar arm:

Hide   Copy Code

```
switch (dicChannel[e.Message.MidiChannel])
{
 ...
 case (int)MIDIInstrument.AcousticGuitarnylon:
 case (int)MIDIInstrument.AcousticGuitarsteel:
 case (int)MIDIInstrument.ElectricGuitarjazz:
 case (int)MIDIInstrument.ElectricGuitarclean:
 case (int)MIDIInstrument.ElectricGuitarmuted:
 case (int)MIDIInstrument.OverdrivenGuitar:
 case (int)MIDIInstrument.DistortionGuitar:
 case (int)MIDIInstrument.GuitarHarmonics:
  this.Dispatcher.Invoke(
    System.Windows.Threading.DispatcherPriority.Normal,
   new Action(
    delegate()
    {
     guitarControl1.Send(e.Message);
    }
   ));
  break;
 ...
```

The strings data are stored in an array of structure named `StringInfo` , which contains info about the 6 strings: minimum and maximum note Ids, the corresponding grid row and the `Rectangle` which represents the visual string on the screen:

Hide   Shrink ▲   Copy Code

```
public partial class GuitarControl : UserControl
{
    ...
    struct StringInfo
    {
        public int Row;
        public int Min;
        public int Max;
        public Rectangle Rect;
    }

    StringInfo[] stringInfos;
    ...
    public GuitarControl()
    {
        InitializeComponent();

        stringInfos = new StringInfo[6];
        stringInfos[0] = new StringInfo()
    { Row = 5, Min = 40, Max = 44, Rect = string5 };
        stringInfos[1] = new StringInfo()
    { Row = 4, Min = 45, Max = 49, Rect = string4 };
        stringInfos[2] = new StringInfo()
    { Row = 3, Min = 50, Max = 54, Rect = string3 };
        stringInfos[3] = new StringInfo()
    { Row = 2, Min = 55, Max = 58, Rect = string2 };
        stringInfos[4] = new StringInfo()
    { Row = 1, Min = 59, Max = 63, Rect = string1 };
        stringInfos[5] = new StringInfo()
    { Row = 0, Min = 64, Max = 90, Rect = string0 };
    }
    ...
```

When a message is sent to the `GuitarControl` , the application checks whether the **`message.Data1`** falls between the minimum and maximum notes:

Hide   Copy Code

```
public void Send(ChannelMessage message)
{
    if (message.Command == ChannelCommand.NoteOn &&
        message.Data1 >= LowNoteID && message.Data1 <= HighNoteID)
    {
```

Next, the application checks **`message.Data2`** . If it is greater than zero, then the string is being pressed. Otherwise, it is being released.

Hide   Copy Code

```
if (message.Data2 > 0)
{
```

Then we see if the `dicNotesOn` dictionary already contains the `message.Data1` . If so, it means the note is already being played, so we don't do anything. Otherwise, we proceed:

Hide   Copy Code

```
if (!dicNotesOn.ContainsKey(message.Data1))
{
```

Next, there's an important part, where we use Linq to select the `StringInfo` structure that matches the note information:

Hide   Copy Code

```
if (!dicNotesOn.ContainsKey(message.Data1))
{
 var row = 0;
 var col = 0;
 var stringId = 0;
 Rectangle stringRect = null;



 var stringInfoQuery = from si in stringInfos
 where message.Data1 >= si.Min && message.Data1 <= si.Max
 select si;

 if (stringInfoQuery.Any())
 {
  var stringInfo = stringInfoQuery.First();
  row = stringInfo.Row;
  col = message.Data1 - stringInfo.Min;
  stringRect = stringInfo.Rect;
  stringId = stringInfo.Row;
 }
```

Then we create the visual elements representing the note being played:

Hide Shrink ▲ Copy Code

```
if (stringRect != null)
{
 stringRect.Stroke =
 stringRect.Fill = stringOnColor;
 stringRect.Height = 1;
}



var noteOn = new Border()
{
 Width = 12,
 Height = 12,
 Background = innerColor,
 BorderBrush = outerColor,
 Tag = stringId,
 CornerRadius = new CornerRadius(2,2,2,2)
};



var txt = new TextBlock()
{
 Text = col.ToString(),
 Foreground = fontBrush,
 HorizontalAlignment = System.Windows.HorizontalAlignment.Center,
 VerticalAlignment = System.Windows.VerticalAlignment.Center,
 FontWeight = FontWeights.Bold,
 FontSize = 10
};

noteOn.Child = txt;

if (stringRect != null)
{
 stringRect.Stroke =
 stringRect.Fill = stringOnColor;
 stringRect.Height = 1;
}

noteOn.SetValue(Grid.RowProperty, row);
noteOn.SetValue(Grid.ColumnProperty, col);
dicNotesOn.Add(message.Data1, noteOn);
grdArm.Children.Add(noteOn);
```

Finally, we also must handle the situations where the note is being released. In this case, we remove the selected `StringInfo` from the `dicNotesOn` dictionary, and remove the visual elements from the `Grid`:

Hide Copy Code

```
    }
    else if (message.Data2 == 0)
    {
        if (dicNotesOn.ContainsKey(message.Data1))
        {
            var noteOff = dicNotesOn[message.Data1];
            dicNotesOn.Remove(message.Data1);
            grdArm.Children.Remove(noteOff);

            var stringId = (int)noteOff.Tag;
            TurnOffString(stringId);
        }
    }
}
else if (message.Command == ChannelCommand.NoteOff)
{
    if (dicNotesOn.ContainsKey(message.Data1))
    {
        var noteOff = dicNotesOn[message.Data1];
        dicNotesOn.Remove(message.Data1);
        grdArm.Children.Remove(noteOff);

        var stringId = (int)noteOff.Tag;
        TurnOffString(stringId);
    }
}
```

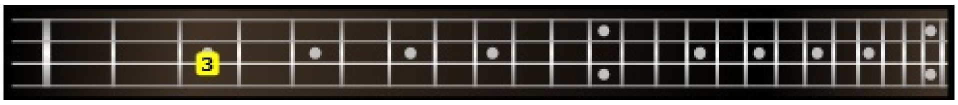
We also have a method for clearing all the notes from dictionary and "turning off" all strings:

Hide   Shrink ▲   Copy Code

```
public void Clear()
    {
        dicNotesOn.Clear();
        for (var i = grdArm.Children.Count - 1; i >= 0; i--)
        {
            if (grdArm.Children[i] is Border)
            {
                var ell = grdArm.Children[i] as Border;
                if (ell.Tag != null)
                {
                    grdArm.Children.RemoveAt(i);
                }
            }
        }

        for (var i = 0; i < 6; i++)
        {
            TurnOffString(i);
        }
    }

private void TurnOffString(int stringId)
    {
        var stringInfo = stringInfos[stringId];
        Rectangle stringRect = null;

        switch (stringId)
        {
            case 0:
                stringRect = string0;
                break;
            case 1:
                stringRect = string1;
                break;
            case 2:
                stringRect = string2;
                break;
            case 3:
                stringRect = string3;
                break;
            case 4:
                stringRect = string4;
                break;
            case 5:
                stringRect = string5;
                break;
        }
        stringRect.Height = 1;
        stringRect.Stroke =
        stringRect.Fill = stringOffColor;
    }
```

The Bass Control

The Bass Control is much like the Guitar Control. The basic differences: it has 2 less strings (the high-pitched ones), and the positions in the instrument's arm correspond to different range of note values.



WPF Midi Band's Bass

Here goes the code that decides if the message should be sent to the `BassControl`:

Hide   Copy Code

```
switch (dicChannel[e.Message.MidiChannel])
{
    ...
    case (int)MIDIInstrument.AcousticBass:
    case (int)MIDIInstrument.ElectricBassfinger:
    case (int)MIDIInstrument.ElectricBasspick:
    case (int)MIDIInstrument.FretlessBass:
    case (int)MIDIInstrument.SlapBass1:
    case (int)MIDIInstrument.SlapBass2:
    case (int)MIDIInstrument.SynthBass1:
    case (int)MIDIInstrument.SynthBass2:
        this.Dispatcher.Invoke(
          System.Windows.Threading.DispatcherPriority.Normal,
            new Action(
                delegate()
                {
                    bassControl1.Send(e.Message);
                }
            ));
        break;
    ...
```

The Drums Control

The Drums Control is probably the most appealing of all. At first, it seemed very difficult to represent a real drum playing in real time. But then an idea came to me, that I could use animations to creating "pulsating" animations for each individual part of the drums. Fortunately, it gave the application a very interesting result.

wpf midi band

WPF Midi Band's Drums

Bass drum, Toms and Snare Drum

Bass Drum	*Floor Tom*	*Tom 1*	*Tom 2*	*Snare Drum*

Here goes a short description of each instrument:*(source: Wikipedia)*

- **Bass Drum**: In music, the bass drum is used to mark or keep time. In marches, it is used to project tempo (marching bands historically march to the beat of the bass). A basic beat for rock and roll has the bass drum played on the first and third beats of a bars of common time, with the snare drum on the second and fourth beats, called "back beats".
- **Floor Tom**: A floor tom is a double-headed tom-tom drum which usually stands on the floor on three legs. However, they can also be attached to a cymbal stand with a drum clamp.
- **Tom-toms**: A wide variety of configurations are commonly available and in use at all levels from advanced student kits upwards. Most toms range in size between 6" and 20", though floor toms can go as large as 24". Two "power" depth tom-toms of 12x10 (12" diameter by 10" depth) and 13x11 is a common hanging tom configuration. Also popular is the "fusion" configuration of 10x8 and either 12x8 or 12x9, and the again popular "classic" configuration of 12x8 and 13x9, which is still used by some jazz drummers. A third hanging tom is often used instead of a floor tom.
- **Snare Drum**: The snare drum is a drum with strands of snares made of curled metal wire, metal cable, plastic cable, or gut cords stretched across the drumhead, typically the bottom. Pipe and tabor and some military snare drums often have a second set of snares on the bottom (internal) side of the top (batter) head to make a "brighter" sound, and the Brazilian caixa commonly has snares on the top of the upper drumhead. The snare drum is considered one of the most important drums of the drum kit.

The 3 classes of drums instruments above were grouped in this section because they share the same kind of `Animations`. Whenever a sound come to those instruments, the drums control triggers an animation on their `ScaleTransform`, so the instruments appear as if they were "pulsating". Of course real drums would never do that, but in the end effect captures well the feeling of the drumbeat.

Besides the "pulsating" animation, these instruments also become more or less transparent when beaten.

Here goes a XAML code example showing how these animations are set up for one instrument:

Hide   Copy Code

```
<UserControl x:Class="WPFMidiHero.Controls.DrumControl"
 ...
    <UserControl.Resources>
 ...
        <Storyboard x:Key="sbSnareDrum" Duration="0:0:0.500">
            <DoubleAnimation From="1.0" To="1.8"
            Storyboard.TargetName="imgSnareDrum"
            Storyboard.TargetProperty="(Grid.RenderTransform).
(ScaleTransform.ScaleX)">
                <DoubleAnimation.EasingFunction>
                    <ElasticEase Oscillations="1" EasingMode="EaseIn" />
                </DoubleAnimation.EasingFunction>
            </DoubleAnimation>
            <DoubleAnimation From="1.0" To="1.8"
            Storyboard.TargetName="imgSnareDrum"
            Storyboard.TargetProperty="(Grid.RenderTransform).
(ScaleTransform.ScaleY)">
                <DoubleAnimation.EasingFunction>
                    <ElasticEase Oscillations="1" EasingMode="EaseIn" />
                </DoubleAnimation.EasingFunction>
            </DoubleAnimation>
            <DoubleAnimation From="0.5" To="1.0"
Storyboard.TargetName="imgSnareDrum"
            Storyboard.TargetProperty="Opacity" AutoReverse="True"/>
 ...
```

Notice in the snippet above that we apply `ElasticEase` function to the animations. This gives up an interesting, more natural effect to the animations.

Since the animations are set in the resources directly in the XAML, when the midi message arrives we must look for the `StoryBoard` corresponding to the correct instrument resource and execute it:

Hide   Shrink ▲   Copy Code

```
public void Send(ChannelMessage message)
{
 if (message.Command == ChannelCommand.NoteOn &&
  message.Data1 >= LowNoteID && message.Data1 <= HighNoteID)
 {
  if (message.Data2>0)
  {
   string sbName = "";
   switch (message.Data1)
   {
    case (int)DrumInstrument.AcousticSnare:
    case (int)DrumInstrument.ElectricSnare:
     sbName = "sbSnareDrum";
     break;
    case (int)DrumInstrument.LowFloorTom:
    case (int)DrumInstrument.HighFloorTom:
     sbName = "sbFloorTom";
     break;
    case (int)DrumInstrument.LowTom:
    case (int)DrumInstrument.LowMidTom:
     sbName = "sbTom1";
     break;
    case (int)DrumInstrument.HiMidtom:
    case (int)DrumInstrument.HighTom:
     sbName = "sbTom2";
     break;
    case (int)DrumInstrument.AcousticBassDrum:
    case (int)DrumInstrument.BassDrum:
     sbName = "sbBassDrum";
     break;
    case (int)DrumInstrument.ClosedHiHat:
    case (int)DrumInstrument.PedalHiHat:
    case (int)DrumInstrument.OpenHiHat:
     sbName = "sbHiHat";
     break;
    case (int)DrumInstrument.CrashCymbal:
    case (int)DrumInstrument.RideCymbal:
     sbName = "sbRideCymbal";
     break;
   }
   if (!string.IsNullOrEmpty(sbName))
   {
    Storyboard sb = (Storyboard)FindResource(sbName);
    sb.Stop();
    sb.Begin();
   }
  }
 }
```

Hi-Hat and Ride Cymbal

Hi-Hat	*Ride Cymbal*

Here goes a short description of each instrument:*(source: Wikipedia)*

- **Hi-Hat**:The hi-hat consists of two cymbals that are mounted on a stand, one on top of the other, and clashed together using a pedal on the stand. A narrow metal shaft or rod runs through both cymbals into a hollow tube and connects to the pedal. The top cymbal is connected to the rod with a clutch, while the bottom cymbal remains stationary resting on the hollow tube. The height of the top-cymbal (open position) is adjustable. When the foot plate of the pedal is pressed, the top cymbal crashes onto the bottom cymbal (closed hi-hat). When released, the top cymbal returns to its original position above the bottom cymbal (open hi-hat). A tension unit controls the amount of pressure required to lower the top cymbal, and how fast it returns to its open position.
- **Ride Cymbal**: The ride cymbal is a type of cymbal that is a standard part of most drum kits. Its function, very similar to the hi-hat it is an alternative to[1], is to maintain a steady rhythmic pattern, sometimes called a ride pattern rather than to provide accents as with, for example, the crash cymbal. The ride can fulfil any function or rhythm the hi-hat does, with the exclusion of an open and closed sound[1]. In rock and popular music another percussion instrument such as a shaker or maraca may be substituted for the cymbal in a ride pattern, especially in quieter styles such as soft-ballads or bossa-nova

These 2 drums instruments have different animations. While the others have that "pulsating" effect, in the Hi-Hat the top cymbal crashes onto the bottom cymbal when the pedal is pressed. This is done by a `DoubleAnimation` targeting the Y coordinate of the `TranslateTransform` of the top cymbal:

Hide   Copy Code

```
<Storyboard x:Key="sbHiHat" Duration="0:0:0.250">
    <DoubleAnimation From="-10.0" To="-35"
    Storyboard.TargetName="imgHiHatTop"
    Storyboard.TargetProperty="(Grid.RenderTransform).
(TranslateTransform.Y)">
        <DoubleAnimation.EasingFunction>
            <ElasticEase Oscillations="1" EasingMode="EaseIn" />
        </DoubleAnimation.EasingFunction>
    </DoubleAnimation>
    <DoubleAnimation From="0.5" To="1.0" Storyboard.TargetName="imgHiHatTop"
    Storyboard.TargetProperty="Opacity" AutoReverse="True"/>
</Storyboard>
```

On the other hand, the Ride Cymbal rotates slightly and quickly 20 degrees:

Hide   Copy Code

```
<DoubleAnimation From="0.0" To="20.0"

    Storyboard.TargetName="imgRideCymbal"

    Storyboard.TargetProperty="(Grid.RenderTransform).
(RotateTransform.Angle)">
        <DoubleAnimation.EasingFunction>
            <ElasticEase Oscillations="1" EasingMode="EaseIn" />
        </DoubleAnimation.EasingFunction>
    </DoubleAnimation>
```

Final Considerations

Thank you for reading **WPF Midi Band**. I hope you have enjoyed it as much as I had. And I also hope the article and the application could be useful for you in some way. And please leave a comment! If you have any complaints, suggestions or doubts, your feedback will be very important, not just for this article, but also for future articles.

History

- 2010-12-31: Initial version
- 2011-01-01: Youtube video added
- 2011-01-02: YouTube videos added, source code changed, Guitar control explained

License

Share

About the Author

Marcelo Ricardo de Oliveira

Instructor / Trainer Alura Cursos Online

Brazil 🇧🇷

Marcelo Ricardo de Oliveira is a senior freelance software developer who lives with his lovely wife Luciana and his little buddy and stepson Kauê in Guarulhos, Brazil, is co-founder of the Brazilian TV Guide TV Map and currently works for Alura Cursos Online.

He is often working with serious, enterprise projects, although in spare time he's trying to write fun Code Project articles involving JavaScript, WPF, Silverlight, XNA, HTML5 canvas, Windows Phone app development, game development and music.

Online Courses (in Portuguese):

Entity LinQ parte 1: Crie queries poderosas em C#
Entity LinQ parte 2: Store Procedures e consultas com o LinQPad
Xamarin parte 1: crie aplicativos mobile com Visual Studio
ASP.NET Core parte 1: Crie uma aplicação MVC com EF Core e Bootstrap
C# Brasil: Formate datas, cpf e números nacionais
C#: Dominando as Collections

You may also be interested in...

Comments and Discussions

Add a Comment or Question
Email Alerts

Spacing Layout Per page

WPf to Flute		Member 14029326	6-Nov-18 19:06

Dears,

Hide Expand ▾ Copy Code

```
I'm Music teacher in Belgium for my pleasure.
I learn the silver flute.

I would like to figure the finger position of Midi songs that a
produce for my pupils and to post that in a video on you tube
as free access for everyone.

I understand with WPF that it is possible to display the midi
notes according a grid position.

I'm not good in programming but I understand variable,
conditions, loops etc.

What I have tried:

I'm not a programmer.
Could you please lead me into your sources to create a new item
"Flute" ?
I can create on my side the .png pictures

In fact it's the opposite figure of the piano.
first position is C, then B, then A, then G, F,E,D,C

If you are interested in, I can send you a mockup via mail.
I figure three rows,

1st row : |X |X |X |X |G#|X |X |X |X |X |
2nd row : |C |X |A |G |X |F |E |D |X |C |
3rd row : |Bb|B |X |X |X |X |X |X |D#|C#|

They are some complication with higher notes but as solution,
each event can be represented by a figure if the displaying is
fast.

Any way, Thank you for your interest and reply.
Kind regard

Jf Denis
mailto:jeanfrancois.denis@belgacom.net
```

Reply·Email·View Thread

Thank you ly45 12-Jun-17 19:28

Fantastic Article	Robert Percy		21-Aug-15 3:20

Definitley Got 5 from me.
Hi Marcelo. I was wondering if you could add a RichTextBox to the form. Using the bass control, every time a note shows on the bass neck, have it write the string and the fret number EX: A5, A7, D4, D5, D7, G4, G6, G7, and so on till the midi file ends. The example is the D-Major Scale. Doing this would make it much easier for bass players to learn more quickly. Also add a copy button so the user can copy and paste the bass tabs to a text file for future reference.

I haven't been programming for awhile and I am very new to WPF.

rspercy75@outlook.com

Reply·Email·View Thread

My vote of 5	Agent__007	21-Jan-15 14:42

Another great read from you! Thank you very much for sharing. 5ed!

Reply·Email·View Thread

My vote of 5	GregoryW	13-Jan-13 7:11

YOU ARE A MASTER !!!! Mazen el Senih 4-Apr-12 1:18

This is just impressive ,
Great idea ,great applying ,Great code and great article !!!

Is there more than 5 I can vote ?
Well shame that does not , but I enjoyed every piece of this article .
Thank you and keep up Mr !

There is always hope ..!

Reply·Email·View Thread

My vote of 5 RC_Sebastien_C 19-Feb-12 23:25

My vote of 5 Wendelius 23-Nov-11 4:39

My vote of 5 Filip D'haene 3-Aug-11 19:22

Re: My vote of 5 Marcelo Ricardo de Oliveira 4-Aug-11 4:12

Thanks man! Glad you enjoyed it 😊
cheers,
marcelo
There's no free lunch. Let's wait for the dinner.

Take a look at Silverlight Menu4U here in The Code Project.

Reply·Email·View Thread

Good example [modified] Gambi11 30-Mar-11 2:00

Does Sanford.Multimedia.Midi.WPF from your solution have any defferences from Sanford.Multimedia.Midi?
modified on Tuesday, March 29, 2011 1:07 PM

Reply·Email·View Thread

Re: Good example	Marcelo Ricardo de Oliveira	31-Mar-11 9:23

Hi Gambi11, thanks a lot!

At that time of the article, I added Sanford.Multimedia.Midi.WPF project to the original Leslie's solution. But it's just throw-away code, it's just a porting of the original Leslie's windows form application to WPF. And it has nothing to do with Sanford.Multimedia.Midi.dll assembly.

Take a look at *Snail Quest* here in The Code Project.

Reply·Email·View Thread

Re: Good example [modified]	Gambi11	1-Apr-11 2:38

Thanks a lot for response.
So I wouldn't lose anything important in my WPF application if I used Sanford.Multimedia.Midi, would I?

modified on Thursday, March 31, 2011 1:44 PM

Reply·Email·View Thread

Re: Good example	Marcelo Ricardo de Oliveira	7-Apr-11 23:42

No, surely not. Take a look at the Snail Quest (see my link below), where I used Sanford.Multimedia.Midi.
BTW, I'd like to know what's your app about 😊
Take a look at *Snail Quest* here in The Code Project.

Reply·Email·View Thread

Re: Good example PeteRainbow 23-Jul-11 15:46

one thing i have found when using the sanford toolkit

is that when in debug mode the performance of loading tracks is terrible

on investigation of this i have found the problem

there is a call in track.cs to AssertValid()

this assert scans the whole track

but on every midi message insert this call is made and so get exponential cost

fix is to just call once at end of loading whole track

i added a boolean to turn on/off

Reply·Email·View Thread

Re: Good example Marcelo Ricardo de Oliveira 24-Jul-11 22:57

Hi Pete, thank you for this info! I'll be working on a new version and that's definitely one of the improvements I'll be working on
best regards,
marcelo
Take a look at Html5 Snooker Club here in The Code Project.

Reply·Email·View Thread

Re: Good example		PeteRainbow	25-Jul-11 1:22

need any help, currently unemployed and trying to keep busy

Reply·Email·View Thread

How to change the speed?		Sina Amirshekari	4-Mar-11 19:26

nice work, thank you
how we can control the speed of playing
?

Reply·Email·View Thread

Re: How to change the speed?	Marcelo Ricardo de Oliveira	31-Mar-11 9:19

Hi Sina, thanks a lot for your feedback.

In order to control the speed, you should investigate Leslie Sanford's C# Midi Toolkit code (follow the link in this article). I'm not sure, but I think it can't be changed in an easy way.

Take a look at *Snail Quest* here in The Code Project.

Reply·Email·View Thread

Re: How to change the speed?	Member 9655439	25-Jun-14 18:13

sequencer1.Clock.PreProcessTempo = 50000;

Reply·View Thread

Re: How to change the speed?	Member 11888107	18-Sep-15 23:38

Nice , very nice	Member 4565433	9-Jan-11 0:05

I come from a DSP audio real-time background, so anything audio is cool by me
And coupled with WPF, double cool

Just love, Moby Dick and Iron Man 😊

Nice

Reply·Email·View Thread

Re: Nice , very nice	Marcelo Ricardo de Oliveira	9-Jan-11 21:45

Thanks a lot for voting! I'm glad you liked it!

cheers,
marcelo

Take a look at *WPF Midi Band* here in The Code Project.

Reply·Email·View Thread

My vote of 5 | GlobX | 8-Jan-11 10:04

I used to play with MIDI in VB6 about a decade ago! Definitely going to get back into it now

Reply·View Thread

Re: My vote of 5 | Marcelo Ricardo de Oliveira | 9-Jan-11 21:44

That's cool! I'm a former VB6 boy and now I'm having a lot of fun with midi again 😊

And thanks for the vote!

cheers,
marcelo

Take a look at *WPF Midi Band* here in The Code Project.