C# MIDI Toolkit

codeproject.com/Articles/6228/C-MIDI-Toolkit

Add your own alternative version

Stats

2.4M views

32.4K downloads

289 bookmarked

Posted 27 Feb 2004

Licenced MIT

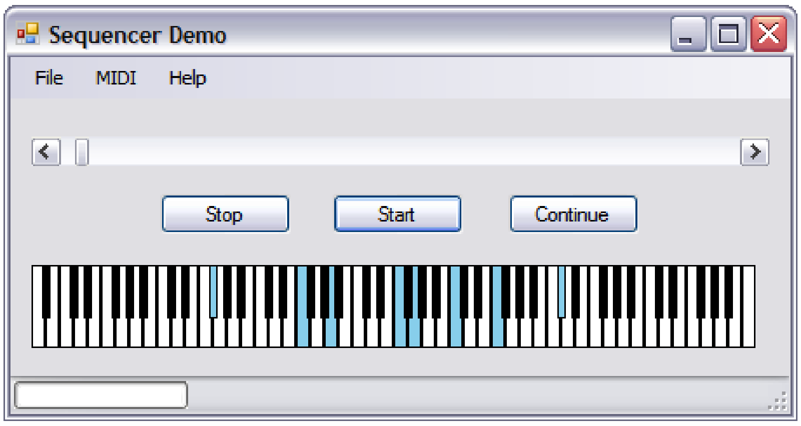
Leslie Sanford, 19 Apr 2007

★★★★★ 4.95 (170 votes)

A toolkit for creating MIDI applications with C#.

- Download source - 59 KB
- Download demo - 202.5 KB

Go to top



Contents

Introduction

This is the fifth version of my .NET MIDI toolkit. I had thought that the previous version was the final one, but I have made many changes that have warranted a new version. This version takes a more traditional C#/.NET approach to flow-based programming, which I'll describe below. I wasn't comfortable with version four's implementation along these lines, so I took a step back and made changes that keep the flow-based approach while remaining within C#/.NET accepted idioms. I'm hoping that this will make the toolkit easier to use and understand.

The toolkit has seen many revisions over the past two to three years. Each revision has been an almost total rewrite of the previous one. When writing software, it is usually a bad idea to make updates that break software using previous versions. However, my goal in creating this toolkit has been to provide the best design possible. As I have grown as a programmer, I have improved my skills and understanding of software design. This has led me to revise the earlier designs of the toolkit without regard to how these revisions will break code. Not exactly the attitude one wants to adopt in a professional setting, but since the toolkit is free and since I have used it as a learning experience to improve my craft, my priorities are different.

top

Flow-based programming

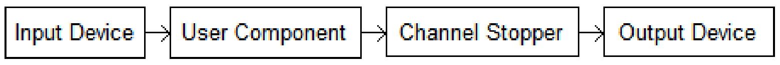
Before I get into the specifics of the toolkit, I would like to talk about its architecture. With each version of the toolkit, I have struggled with how to structure the flow of messages through the system. I wanted an approach that would be versatile and allow customization. It would be nice, I thought, if users could plug their own classes into the flow of MIDI messages to do whatever they want. For example, say you wanted to write a harmonizer, a class capable of transposing notes in a specified key to create harmony parts automatically. It should be easy to simply plug the harmonizer into the toolkit without affecting other classes. In other words, the toolkit should be customizable and configurable.

Investigating this problem led me to J. Paul Morrison's excellent website on flow-based programming. He has written a book on the subject, which can be found on his website as well as at Amazon.

The idea is simple and will probably seem familiar to most: Data flows through a network of components. Each component can do something interesting with the data before passing it along to the next component. In design pattern terms, this approach is most like the Pipe and Filter pattern and is also similar to the Chain of Responsibility pattern. Please check out J. Paul Morrison's excellent book for more information.

(Just to be clear: when I say "component," I'm not necessarily talking about classes that implement the `IComponent` interface. I'm speaking in more general terms. A component is simply an object in a chain of objects designed to process the flow of messages.)

Below is a very basic network of components designed to handle the flow of MIDI channel messages:



The flow of messages begins with the input device. An input device receives MIDI messages from an external source. Next, the messages flow through a user component. This component might want to do something like change the MIDI channel, transpose note messages, or change the messages in some way. Then the messages pass through the channel stopper. This component simply keeps track of all currently sounding notes.

When the message flow stops, the channel stopper can turn off all sounding notes so that none of them hang. Finally, the messages reach the output device. Here they are sent to an external MIDI device.

top

Implementing flow-based programming in C#

Well, this is something I really struggled with. You can read a bit about the different ways I tried to achieve this by reading my blog. I found myself going round in circles on this. In version four of the toolkit, I settled on the idea of using a source/sink abstraction. I created an interface representing "Sources." A source represents a source of MIDI messages. "Sinks" were represented by delegates that could be connected to sources; a sink is simply a method capable of receiving a MIDI message. This worked well but it was a little confusing because the implementation looked a little "funny." That is to say, a C# programmer looking at the code for the first time might be confused as to what is going on.

I decided to do away with the sink/source infrastructure and use something more idiomatic. Sources of MIDI messages raise events when they have messages to send. Instead of implementing an interface and having `Connect` and `Disconnect` methods for hooking to sinks, they would simply have events. There are two advantages here: First, sources no longer have to implement an `ISource` interface, and second, .NET events are something very familiar to us. So sources of MIDI messages now look like your everyday class that just happens to have one or more events.

How about sinks, those objects that can *receive* MIDI messages? A sink can be anything. It's just an object that has a method that can receive a MIDI message. In version four, I had a `Sink` delegate for representing methods of objects capable of receiving MIDI messages. These delegates were used to connect with sources. This approach of using delegates to "connect" sources and sinks is still used in the toolkit but not as before. Instead, delegates are used as adaptors that connect to the events raised in sources and adapts the events so that objects that need to receive the messages can do so without any knowledge of the source.

Let's look at an example. Say that we're using an `InputDevice` to receive MIDI messages from a MIDI device, such as your soundcard. The `InputDevice` raises a `ChannelMessageReceived` event each time it receives a channel message. Suppose that we want to keep track of any note-on channel messages so that when we decide to stop receiving messages, we can turn off any currently sounding notes to keep them from "hanging." The `ChannelStopper` class is just for this purpose. However, the `ChannelStopper` has no knowledge of the `InputDevice` class. We need a way to hook them up so that messages generated by the `InputDevice` can be passed along to the `ChannelStopper`. Here is how we can do this with an anonymous method:

Hide Copy Code

```
InputDevice inDevice = new InputDevice(0);
ChannelStopper stopper = new ChannelStopper();

inDevice.ChannelMessageReceived += delegate(object sender,
ChannelMessageEventArgs e)
{
    stopper.Process(e.Message);
};

inDevice.StartRecording();
```

In this example, an anonymous method adapts events raised by the `InputDevice` so that they can be processed by a `ChannelStopper`. The `InputDevice` is the source of channel messages and the `ChannelStopper` is a sink capable of receiving and processing channel messages. The nice thing about this approach is that no explicit source/sink infrastructure is needed. Neither class knows anything about being a source or sink. The flow of messages is orchestrated by an external agent, in this case, an anonymous method.

top

MIDI messages

There are several categories of MIDI messages: Channel, System Exclusive, Meta, etc. In designing a MIDI toolkit, the challenge is to decide how to represent these messages. One approach is to create two or three general MIDI classes and have specific types of MIDI messages represented through the properties of those classes. The Java MIDI API takes this route. Another approach is to create a large collection of finely grained classes to represent all of the different types of MIDI messages. For example, there are many types of Channel messages such as the Note-on, Note-off, Program change, and Pitch Bend messages. The fine grained approach would create a class for each of those message types. My approach was to take a middle ground. I created classes for the general categories of MIDI messages but left the specific types of messages as properties within those classes. This kept the class hierarchy lightweight and manageable while providing enough specialization to make working with MIDI messages easy.

Here is the hierarchy of MIDI message classes in the MIDI toolkit:

`IMidiMessage`
- `ShortMessage`
    - `ChannelMessage`
    - `SysCommonMessage`
    - `SysRealtimeMessage`
- `SysExMessage`
- `MetaMessage`

Specific types of messages are represented through properties. For example, the `ChannelMessage` class has a `Command` property that can be set to represent the various types of Channel messages.

top

Message builders

All message classes are immutable. This makes sharing messages throughout an application safe. To create messages, you pass the desired property values to their constructor. Additionally, the toolkit provides a set of builder classes for making message creation more convenient.

The toolkit provides the following message builders:

- `ChannelMessageBuilder`
- `SysCommonMessageBuilder`
- `KeySignatureBuilder`
- `MetaTextBuilder`
- `SongPositionPointerBuilder`
- `TempoChangeBuilder`
- `TimeSignatureBuilder`

The `ChannelMessageBuilder` and the `SysCommonBuilder` also use the Flyweight design pattern. When a new message is built, it is stored in a cache. When another message is needed that has the exact same properties as a message that has already been built, the previous message is retrieved rather than creating a new one. When you consider that a typical MIDI sequence is made up of thousands of messages, many of them identical, it is easy to see how the Flyweight pattern is applicable.

Here is an example of creating a `ChannelMessage` object representing a note-on message:

Hide Copy Code

```
ChannelMessageBuilder builder = new ChannelMessageBuilder();

builder.Command = ChannelCommand.NoteOn;
builder.MidiChannel = 0;
builder.Data1 = 60;
builder.Data2 = 127;
builder.Build();
Console.WriteLine(builder.Result);
```

After the builder has been initialized with the desired properties, the MIDI message is built with a call to the `Build` method. The MIDI message can then be retrieved via the `Result` property.

There are several builder classes for creating specific types of meta messages. For example, to create a key signature meta message, you use the `KeySignatureBuilder` class:

Hide Copy Code

```
KeySignatureBuilder builder = new KeySignatureBuilder();
builder.Key = Key.CMajor;
builder.Build();
Console.WriteLine(builder.Result);
```

top

MessageDispatcher class

Often there is a need to process a collection of `IMidiMessage` s. How each message is processed depends on its type. The problem is that you cannot tell an `IMidiMessage`'s type without an explicit check. The `IMidiMessage` provides a `MessageType` property just for this purpose. However, having to repeatedly check message types throughout your code can be cumbersome.

The `MessageDispatch` class is designed to automate these checks. This class acts as a source for every type of MIDI message. It raises an event each time it dispatches a message. The type of event is determined by the type of message it is dispatching.

top

Clocks

MIDI playback is driven by ticks that occur periodically. The source of these ticks are MIDI clocks. MIDI clocks come in all shapes and sizes. For example, playback can be driven by an internal or external clock. Also, the way in which the ticks are generated depends on whether the MIDI sequence has pulses per quarter note resolution or SMPTE resolution. For the vast majority of situations, an internal clock generating ticks with pulses per quarter note resolution is all you need.

The `IClock` interface represents the basic functionality for all MIDI clocks:

Hide   Copy Code

```
public interface IClock
{
    event EventHandler Tick;
    event EventHandler Started;
    event EventHandler Continued;
    event EventHandler Stopped;
    bool IsRunning
    {
        get;
    }
}
```

The `Tick` event occurs when a MIDI tick has elapsed. The `Started`, `Continued`, and `Stopped` events are self-explanatory. However, it should be pointed out that when the `Started` event occurs, sequence playback starts from the beginning of the sequence. When the `Continued` event occurs, playback starts from the current position. The `IsRunning` property indicates whether the clock is running.

You may notice that there are no methods in the interface for starting and stopping a clock. That is because with clocks that are driven by an external source, the source is responsible for starting and stopping the clock. The clocks receive messages via MIDI and based on those messages starts or stops generating ticks. Since all MIDI clocks implement `IClock`, it only represents the functionality common to all the clocks.

At this time, the toolkit provides only one clock class, the `MidiInternalClock`. This clock generates MIDI ticks internally using pulses per quarter note resolution. For the majority of situations, this clock will work fine.

The `MidiInternalClock` has a `Tempo` property for setting the tempo in microseconds per beat. To set the tempo to 120 bpm, for example, you would set the `Tempo` property to 500000. It can receive meta tempo change messages. When a meta tempo change message is passed to it, it changes its tempo to match the tempo represented by the message.

top

MidiEvent class

A MIDI file is made up of several tracks. Each track contains one or more timestamped MIDI messages. The timestamp represents the number of ticks since the last message was played. This timestamp is called delta ticks. The `MidiEvent` class represents a timestamped MIDI message. It has three `public` properties:

- `DeltaTicks`
- `AbsoluteTicks`
- `MidiMessage`

The `DeltaTicks` property represents the number of ticks since the last `MidiEvent`. In other words, this value represents how long to wait after playing the previous `MidiEvent` before playing the current `MidiEvent`. For example, if the `DeltaTicks` value is 10, we would allow 10 ticks to elapse before playing the MIDI message represented by the current `MidiEvent`.

The `AbsoluteTicks` represents the overall position of the `MidiEvent`. This is the total number of ticks that have elapsed until the current `MidiEvent`.

The `MidiMessage` property is the `IMidiMessage` represented by the `MidiEvent`.

In addition there are two internal properties, one which points to the previous `MidiEvent` in the track, and one which points to the next `MidiEvent` in the track. In other words, the `MidiEvent` class acts as a node in a doubly linked list of `MidiEvent`s.

top

Track class

The `Track` class represents a collection of `MidiEvent`s. It is responsible for maintaining a collection of `MidiEvent`s in proper order. `MidiEvent`s are not directly added to a `Track`. Instead, you add an `IMidiMessage`, specifying its absolute position in the `Track`. The `Track` then creates a `MidiEvent` to represent the message and inserts it into its collection of `MidiEvent`s.

In addition to providing functionality for adding and removing MIDI events, the `Track` class also provides several iterators. There is a standard iterator that simply iterates over the `MidiEvent`s one at a time. Another iterator takes a

`MessageDispatcher` object and passes each `IMidiMessage` to the dispatcher which in turn raises an event specific to the type of message it is dispatching. The value the iterator returns is the absolute ticks for the current `MidiEvent`.

Perhaps the most useful iterator is the one that when advanced moves forward only one tick at a time. The iterator keeps track of its tick position in the `Track`. When the tick count has reached a value in which it is time to play the next `MidiEvent`, it passes the `IMidiMessage` represented by the `MidiEvent` to the `MessageDispatcher` and returns the absolute tick count. This iterator also takes a `ChannelChaser` object as well as a start position value and "chases" up to the start position before switching to the playback mode. In essence, this iterator allows us to stream the `Track` in real-time.

top

Sequence class

The `Sequence` class represents a collection of `Track`s. It also provides functionality for loading and saving MIDI files, so `Sequence`s can load and save themselves.

Every `Sequence` has a division value. This value represents the resolution of the `Sequence` and is represented by a property. There are two types of division values: Pulses per quarter note and SMPTE. The `Sequence` has a `SequenceType` property indicating the sequence type. Unfortunately, SMPTE sequences aren't supported at this time.

top

MIDI devices

There are several MIDI device classes in the toolkit. Each device class is derived directly or indirectly from the abstract `Device` class in the `Sanford.Multimedia` namespace. The `InputDevice` class represents a MIDI device capable of receiving MIDI messages from an external source, such as a MIDI keyboard controller or synthesizer. The `OutputDeviceBase` class is an `abstract` class that serves as the base class for the output device classes. The `OutputDevice` class represents a MIDI device capable of sending MIDI messages to an external source or your soundcard. And the `OutputStream` class encapsulates the Windows Multimedia MIDI output stream API. It is capable of playing back timestamped MIDI messages.

There can be more than one of these devices present on your computer. To determine the number of input devices present, for example, you would query the `InputDevice`'s `static DeviceCount` property. The output device classes also have this property.

Each MIDI device has its own unique ID. This is simply an integer value representing the device's order in the list of devices available. For example, the first output device on your system would have an ID of 0. The second output device would have an ID of 1, and so on. The same is true for the input devices. When you create a MIDI device, you pass it the ID of the device you wish to use to its constructor. If there was an error in opening the device, an exception is thrown.

To find out the capabilities of a device, you query the class' `static` `GetDeviceCapabilities` method, passing it the device ID of the device you are interested in. This method will return a structure filled with values representing the capabilities of the specified MIDI device.

Let's describe each device class in detail:

top

InputDevice class

The `InputDevice` class represents a MIDI device capable of receiving MIDI messages. It has an event for each of the major MIDI message it can receive. To receive MIDI messages, you connect to one or more of these events. Then you call the `StartRecording` method. Recording will continue until either `StopRecording` or `Reset` is called. The `InputDevice` lets you set the size of the sysex buffer it uses to receive sysex messages. When the `InputDevice` has received a complete sysex message, it raises the SysExReceived event.

top

OutputDeviceBase class

The `OutputDeviceBase` class is an `abstract` class that provides basic functionality for sending MIDI messages. It has several overloaded `Send` methods for sending various types of MIDI messages.

top

OutputDevice class

The `OutputDevice` class represents a MIDI device capable of sending MIDI messages. It inherits most of its functionality from the `OutputDeviceBase` class. It also provides running status functionality.

The following code creates an `OutputDevice`, builds and sends a note-on message, sleeps for one second, and then builds and sends a note-off message:

Hide Copy Code

```
using(OutputDevice outDevice = new OutputDevice(0))
{
    ChannelMessageBuilder builder = new ChannelMessageBuilder();

    builder.Command = ChannelCommand.NoteOn;
    builder.MidiChannel = 0;
    builder.Data1 = 60;
    builder.Data2 = 127;
    builder.Build();

    outDevice.Send(builder.Result);

    Thread.Sleep(1000);

    builder.Command = ChannelCommand.NoteOff;
    builder.Data2 = 0;
    builder.Build();

    outDevice.Send(builder.Result);
}
```

top

OutputStream class

The `OutputStream` class is also derived from the `OutputDeviceBase` class. It encapsulates the Windows multimedia MIDI output stream API. It provides functionality for playing back MIDI messages.

To play MIDI messages, you call `StartPlaying`. The `OutputStream` will then begin playing back any MIDI messages in its queue. To place MIDI messages in the queue, you first write one or more `MidiEvent`s using the `Write` method. After writing the desired number of `MidiEvent`s, you call `Flush`. This flushes the events to the stream causing it to play them back.

top

Sequencer Class

The `Sequencer` class is back. It's a lightweight class for playing back `Sequence`s. I felt the previous `MidiFilePlayer` class was not the best means for playing back MIDI sequences. I wanted to give the toolkit the ability to play `Sequence`s your create programmatically. One issue that caused me to shy away from a `Sequencer` class (after having created one in earlier versions) is the problem of a `Sequence` changing as it's being played by a `Sequencer`. I still haven't solved that problem, but I didn't want that issue to prevent easy `Sequence` playback. So I'm putting in a new version of the `Sequencer` class with the understanding that it's meant to be used for simple playback. For something more sophisticated, you can use it as the basis for creating something more.

top

Dependencies

The MIDI toolkit depends on the `DelegateQueue` class from my `Sanford.Threading` namespace; the `InputDevice` and `OutputDevice` classes use it for queueing MIDI events. In turn, the `Sanford.Threading` namespace depends on my `Sanford.Collection` namespace, so that assembly is also necessary for the toolkit to compile. Finally, the toolkit uses the `Sanford.Multimedia` namespace. I've provided all of the assemblies with the download. I've linked the projects that use them to the assemblies in hopes that the toolkit will compile out of the box. Hopefully, the days of having trouble compiling my projects because of not having the right assemblies are over.

top

Conclusion

This article has provided an overview of my .NET MIDI toolkit. My hope is that it will be a useful and powerful tool for writing MIDI applications. It has been a lot of fun to write. Each version has represented the very best of my skill and knowledge as a programmer. I welcome feedback and any bug reports you may have. Take care and thanks for your time.

top

History

- 20th Feb, 2004
  Article submitted.
- 07th May, 2004
  Second major version.
- 27th Oct, 2004
  Third major version.
- 08th March, 2006
  Fourth major version.
- 14th March, 2006
  - Cleaned up the article a bit.
  - Consolidated the sources into one generic interface in the `Multimedia` namespace.
  - Consolidated all of the sinks into one generic delegate also in the `Multimedia` namespace.
  - Fixed a bug in the MIDI File Player demo as well as a bug in the `Sequence` class.
- 14th April, 2007
  Fifth major version.

top

License

Share

About the Author

Leslie Sanford

United States

Aside from dabbling in BASIC on his old Atari 1040ST years ago, Leslie's programming experience didn't really begin until he discovered the Internet in the late 90s. There he found a treasure trove of information about two of his favorite interests: MIDI and sound synthesis.

After spending a good deal of time calculating formulas he found on the Internet for creating new sounds by hand, he decided that an easier way would be to program the computer to do the work for him. This led him to learn C. He discovered that beyond using programming as a tool for synthesizing sound, he loved programming in and of itself.

Eventually he taught himself C++ and C#, and along the way he immersed himself in the ideas of object oriented programming. Like many of us, he gotten bitten by the design patterns bug and a copy of GOF is never far from his hands.

Now his primary interest is in creating a complete MIDI toolkit using the C# language. He hopes to create something that will become an indispensable tool for those wanting to write MIDI applications for the .NET framework.

Besides programming, his other interests are photography and playing his Les Paul guitars.

You may also be interested in...

Comments and Discussions

Last Visit: 28-Feb-19 13:33 Refresh 1234567891011
Last Update: 28-Feb-19 13:33 Next »

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.