teVirtualMIDI-SDK (1.3.0.43)

Disclaimer & legal stuff

About teVirtualMIDI

teVirtualMIDI is a virtual MIDI driver for Windows 7 up to Windows 10 for both 32bit and 64bit. The teVirtualMIDI driver is a special MIDI driver that is not bound to any specific hardware.

The driver can create MIDI-ports on the system, that are accessible like every other hardware bound driver by applications - either by the Multi-Media-Api, Direct-Music or Kernel-Streaming.

The driver can instantiate up to 256 virtual ports on Windows 7, Windows 8 and Windows 10 (also on secure boot systems)

The driver is multi-client-capable. This means several applications can use the public port at the same time.

This driver allows Windows applications and services to create freely nameable MIDI-ports on the fly, a functionality that already exists on other Operating-systems but unfortunately is missing from Windows.

Instead of talking to some hardware device via PCI(e), USB, Firewire etc. this driver has a private backend that is exposed over a simple-to-use interface-dll. This backend is used to dynamically create & destroy the freely nameable ports and send/receive data to/from the public exposed end.

As the driver is used by popular tools rtpMIDI (network-MIDI-driver) and loopMIDI (loopback-MIDI-cable) it is in use by many thousands of users every day and this already for a couple of years. Therefore the driver has a high level of field-proven features & stability.

About the teVirtualMIDI-SDK

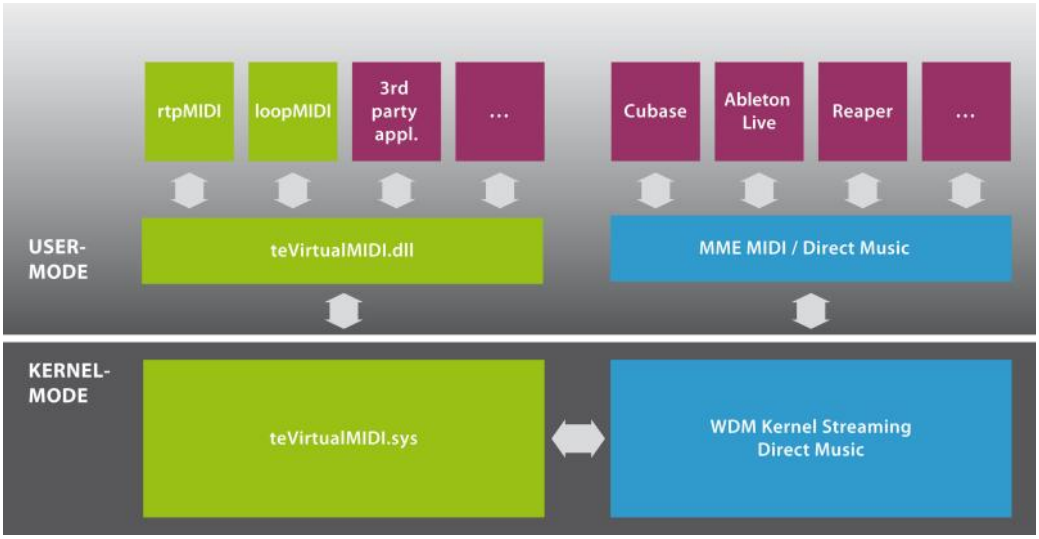
With the teVirtualMIDI-SDK an application can easily integrate the ability to create virtual MIDI-ports on the fly, even with ability to specify a custom-name for the designated MIDI-port.

This SDK contains bindings for C / Delphi and class-wrappers for .NET & Java. Also included are command line sample-applications that use the bindings/class-wrappers to implement a simple single loopback-midi-port. An installer MSI is available for licensees for integration into their product setup.

Application examples:

- A stand-alone virtual Instrument wants to create a virtual Port, so sequencer applications can access it.
- Use special devices like HID-devices, Kinect etc. to create MIDI-data and connect to other applications without the need for loopback MIDI drivers
- Create non-standard transports for MIDI like over the network (an example would be the rtpMIDI driver which uses the teVirtualMIDI driver to create the local-end of the network MIDI-ports)
- Create MIDI-filter applications

Architecture



The driver is made up of two main components:

- The driver
  implements a DMus miniport driver that integrates into the Microsoft Audio/MIDI-system. Therefore is highly compatible to any MIDI-applications. The driver has a private backend which is used to create the ports on the public end and transfers data between the public interface and the application which created the port via the private interface.

- The interface-DLL
  implements the application-interface to the driver and provides a simple, easy-to-use API. This “raw” interface can be observed in the C and Delphi-bindings. With just 4 functions ports can be created, destroyed, MIDI-commands be sent or received.

The interface-DLL is available for both 32bit & 64bit. So you can write your application in both. The public interface is compatible to both 32bit and 64bit as well. So even if your application is 32bit, the application using the public port can be 64bit (and vice versa).

Depending on the operating-system, teVirtualMIDI is able to create up to 256 virtual MIDI-ports at the same time. Bear in mind that all applications using this driver "consume" this resource. So if you already use rtpMIDI with 10 ports and loopMIDI with another 15, your application will have to do with the remaining 231!

teVirtualMIDI is also multi-client-capable. This means that several applications can open one of your virtual Ports at the same time (the limit for concurrent applications is 8 at this time). This is totally transparent for your application. teVirtualMIDI takes all the MIDI-data that is sent from those multiple applications and merges the MIDI-commands into one stream that you receive on your private interface. All MIDI-data that you send via the private interface is sent to all the applications that have opened your virtual port.

You can almost freely specify a name for the virtual MIDI-port you want to create. It is limited to 31 characters in length. Otherwise applications using the Microsoft Multimedia Extensions for MIDI-operation will not be able to see the complete name of your port. Certain characters are also "off-limit": ("\", "/", "<", ">", """, ":","|","?","*"). Other than that feel free to use the Unicode-character set...

Software that used this SDK to implement virtual MIDI ports, needs an installed teVirtualMIDI driver. For licensees of this SDK, there exists a MSI-module for explicit installation of the driver.

For testing and internal use it is possible to just install either loopMIDI or rtpMIDI. The installation of both of those tools will implicitly also install the teVirtualMIDI driver and thus enables the usage by your application.

C/C++ Binding

The C/C++ binding consists of a header-file which specifies the API-entries of the interface-DLL. Also included are 32bit and 64bit lib-files to link the DLL to your application. If you do not wish to be dependent on the installation of the DLL on your target-computer, just get the entry-points "by hand" using LoadLibrary and GetProcAdress.

Included is also a simple C command-line sample-program that implements a single loopback-MIDI-port and dumps all the commands it forwards between input & output on the console as hex-strings.

You find a good amount of other useful information about the usage of the interface-DLL in the header file itself.

Simple usage:

- Create port using virtualMIDICreatePortEx2
  You specify
  - the desired name of your virtual port.
  - if you want to use callbacks, a pointer to your callback-function. Otherwise you specify NULL
  - some user-specific pointer/data that will be referenced in your callback (you can use this to reference your port-instance-specific data like a pointer to a C struct or a C++ class for example).
  - the maximum size of Sysex-data that you want to receive.
  - and the operational flags (currently only one flag exists to specify if you want to receive preparsed MIDI-commands or an unparsed stream that you may parse yourself)

  You will receive a pointer to the port that you have just created. In case you receive a result of NULL, the call has failed and you can use GetLastError() to retrieve the cause of this failure.

- Send MIDI-commands via call to virtualMIDISendData.
  Here you specify
  - the pointer that you have gotten as a result of virtualMIDICreatePortEx2
  - a pointer to one fully valid MIDI-command in memory
  - the length of the MIDI-command

  You will receive a BOOL value as a result. If the data could be sent, the return value will be TRUE, otherwise call GetLastError() to get the reason of the failure.

- Handle calls to your callback
  If you have specified the flag to get parsed MIDI-commands, you will receive one callback for each MIDI-command received.
  The data in the parameters you receive will be
  - The pointer to the instance of the port that you received in virtualMIDICreatePortEx2
  - A pointer to the MIDI-command that you need to handle
  - The length of the MIDI-command that you need to handle
  - The user-specific pointer/data that you specified in your call to virtualMIDICreatePortEx2

- Close your created port using virtualMIDIClosePort
  Again you specify
  - the pointer that you have gotten as a result of virtualMIDICreatePortEx2

With just those three API-calls and one callback, you have all that is necessary to use virtual MIDI-ports with this SDK!

Advanced topics

In C/C++ (as in Delphi), you have two choices of operation:
- Blocking Receive
- Callback

The sample is implemented using callbacks, so it does not need to handle a separate thread (actually the interface-DLL just takes over this task, so there still is an additional thread involved).

Bear in mind, that the callbacks you will receive from the interface-dll will be in arbitrary thread-context. So if you access other software-packages from this callback that rely on things like TLS etc. you are responsible for setting up the correct environment.

If you opt to use virtualMIDIGetData instead of a callback (in this case you specify NULL as the parameter for the callback in virtualMIDICreatePortEx2), you should be calling this in your own thread, since this function blocks until it either receives a MIDI-command, or returns with an error-code specifying that something went amiss or the driver-instance has been shut down.

You can call virtualMIDIShutdown from you main thread to abort an ongoing virtualMIDIGetData. Afterwards, you wait for the termination of the thread that called the virtualMIDIGetData and then close the port using virtualMIDIClose.

Apart from the decision: Blocking Receive/Callback, I assume that you are familiar with multi-threaded development and the necessary precautions like locking to avoid race-conditions and keep locking-order in the right way to stay away from deadlocks…

Whenever an operation failed, the interface-DLL will set the last-error in the local thread environment. You can retrieve that with the GetLastError() system-call. Some of the more specific error-codes are listed in the header-file. A more complete list can be found in the Delphi-Code which maps those codes to error-strings.

You may not call virtualMIDISendData multiple times at once – you need to wait until the first call returns. The same is true for virtualMIDIGetData. If you do, you will receive an error-code for the seconds call of the functions.

During development, you might also take a look at the function virtualMIDILogging. Depending on the specified logging-flags, various internal information of the interface-DLL is printed with OutputDebugString. Those outputs can either be caught using your normal debugger, or when running your application standalone, just use the tool DbgView.exe which can be downloaded from the Microsoft website.

Delphi Binding

The Delphi binding consists of a small unit that wraps the API-entries in Delphi function calls and also includes a mapping function to convert possible error-results in human-readable form.

This unit explicitly tries to load the interface-dll. This has the advantage, that an application that is only optionally using teVirtualMIDI will not fail to load when the driver and it's interface-DLLs are not installed on the system.

The wrapper also tracks usage of open ports and unloads the interface-DLL if no ports are currently in use. In case some other installer tries to install a new version of the teVirtualMIDI-driver, the installer will shut down the driver first. If you have any ports in use, you will receive an error telling you that the driver is not working anymore. If you then close all of your ports, the interface-DLL will not be locked anymore and the driver-install will go through without the need of a reboot…

Like the C/C++ wrapper also the Delphi binding contains a simple loopback-MIDI command-line application.

Apart from the added functionality in Delphi, the topics above concerning the C/C++ binding apply to Delphi as well – so if you take a look at the section "Simple usage", this is a template on exactly how the SDK can be used in Delphi as well.

.NET Class

For the usage of teVirtualMIDI in .NET, I have created a small C# class that wraps the drivers functionality.

Since using callbacks from native-code into managed code is a pain, I have only provided the ability to use the blocking function to get data from

First you create an instance of the TeVirtualMIDI class. You need to specify at least the name of the port you want to create. If you do not specify a value for Sysex-size or flags, default-values will be used.

Afterwards you can call the instance-method "sendCommand" with a byte-array that contains one valid MIDI-command.

For retrieving MIDI-commands, just call the instance-method "getCommand" which will block until a valid MIDI-command is available. Your result will be a byte-array containing one valid pre-parsed MIDI-command. Since this method is blocking, you should be calling it from within a worker-thread.

When you are done using your virtual MIDI port, just call the instance-method "shutdown". If there is a "getCommand" currently in progress it will abort.

It is important that you call the "shutdown" method, otherwise the thread will never terminate and the port will not be removed from the system since the garbage-collection will not recollect it…

Any of the .NET operations might throw a TeVirtualMIDIException. For common things, plain text-exception-messages have been defined. You can also retrieve the actual Windows-system-error that had been retrieved within the class using “GetLastError()”, using the reasonCode property.

Like done in the C/C++ and Delphi bindings, I have created a small sample loopback command-line tool in C# as well.

Java Class

Likewise to the .NET-wrapper, I have created a small Java class that wraps the drivers functionality. If you compare the code of the two they are pretty identical and all of the remarks in the .NET section apply here as well.

Since Java needs a JNI-dll to access native DLLs, I have integrated those Java-specific entry-points right into the interface-DLL. So instead of having a separate JNI-dll referring to the interface-DLL, all has been integrated into one single DLL.

NOTE: The initial version of the SDK did not place the TeVirtualMIDI and TeVirtualMIDIException classes into a package. This has now been fixed. For reasons of backwards-compatibility, the old JNI-entry-points are still existing!

As with C/C++, Delphi and .NET again here is a sample loopback command-line tool included.

Unidirectional ports

Since driver-version 1.2.0, teVirtualMIDI also supports the instantiation of unidirectional virtual MIDI-ports.

If you specify "TE_VM_FLAGS_INSTANTIATE_RX_ONLY" in the flags-part of virtualMIDICreatePortEx2, you will create a unidirectional MIDI-out port. Therefore, no MIDI-in device will be visible to applications using the public end of your created port. If you have instantiated the virtual port this way, calls to virtualMIDISendData will obviously fail.

If you specify "TE_VM_FLAGS_INSTANTIATE_TX_ONLY" in the flags-part of virtualMIDICreatePortEx2, you will create a unidirectional MIDI-in port. Therefore, no MIDI-out device will be visible to applications using the public end of your created port. If you have instantiated the virtual port this way, calls to virtualMIDIGetData will obviously fail. Also you will not receive any MIDI-data if you specified a callback for the announcement of incoming MIDI-data.

If you specify both "TE_VM_FLAGS_INSTANTIATE_RX_ONLY" and "TE_VM_FLAGS_INSTANTIATE_TX_ONLY" or "TE_VM_FLAGS_INSTANTIATE_BOTH", or none of those, you will get the "normal" behaviour of a bi-directional port.

Manufacturer & Product GUIDs

Also since driver-version 1.2.0, teVirtualMIDI supports the ability to specify a manufacturer and product GUID when instantiating a port using teVirtualMIDICreatePortEx3. The GUIDs specified there will be visible to applications using midiInGetDevCaps and midiOutGetDevCaps when using the extended MIDIINCAPS2 and MIDIOUTCAPS2 structures.

This feature can be used to uniquely identify user-created virtual MIDI-ports from within applications.

If NULL is specified for those parameters or when using one of the older teVirtualMIDICreatePortXXX functions, the created port will get the default GUIDs provided by the teVirtualMIDI-driver:

Manufacturer: Tobias Erichsen - {29A439DC-5E9B-40ec-89A5-C3AB12241ED6}
Product: teVirtualMIDI - {9FDAF55E-5925-4d14-9073-531309729363}

Retrieving Process Ids

Since driver-version 1.2.7, teVirtualMIDI supports the ability to retrieve the Windows process ids of the applications that are using the public interface of the created virtual ports. Using this feature you can either make sure not to destroy a port before all applications have released their use of the port, or you could show the actual applications in your GUI. For this functionality you simply call virtualMIDIGetProcesses, providing a buffer into which the respective process ids are written.