Revista Vórtex | Vortex Music Journal | ISSN 2317–9937 | http://vortex.unespar.edu.br/
D.O.I.: https://doi.org/10.33871/23179937.2021.9.2.20

[vstplugin~] – A Pd external for hosting VST plugins

Christof Ressi

Anton Bruckner Privatuniversität | Austria

Abstract: [vstplugin~] is a cross-platform Pure Data external for hosting VST 2 and VST 3 plugins that has been developed at the IEM in Graz. The external provides an extensive set of features which span several topics, such as channel layout, plugin search, real-time safety, GUI editor, parameter automation, preset management, MIDI events, transport and timing, bridging and sandboxing, multithreading and offline processing. We show the development process and discuss previous work in this area. We also cover the history of the VST technology, explain the differences between the VST 2 and VST 3 SDK and offer insight into their inner workings. Finally, we discuss possible improvements and extensions for the future.

Keywords: Pure Data, IEM, VST plugin, Pd external

Received on July 1st, 2021. Approved on: November 1st, 2021. Available online on: 30/11/2021. Guest Editor: Dr. Alexandre Torres Porres.

The [vstplugin~] external allows to host VST 2 and VST 3 plugins in Pure Data on Windows, macOS and Linux. It is part of the *vstplugin* project (Ressi 2021) which also includes an extension for SuperCollider. [vstplugin~] is not the first object of its kind: between 2003 and 2005, Thomas Grill developed the [vst~] Pd external (Grill 2009), which itself is based on the work of Jarno Seppänen and Mark Williamson. However, [vst~] has certain limitations. For example, it does not allow to set or get the internal plugin state, which means you cannot save and load presets. Also, it is only available for Windows and macOS. Finally, it does not support VST 3 plugins, which can be easily explained by the fact that the VST 3 SDK has only been introduced in 2006.

Since VST 3 support has become increasingly important (which will be explained in the next section), I essentially needed a common wrapper interface which would abstract away the vast differences between the VST 2 and VST 3 SDK. I could have used a framework like JUCE, but I tried to avoid external dependencies as much as possible. Also, I needed absolute control over the wrapper to meet the specific needs of Pure Data and SuperCollider. Eventually, I decided to write everything from scratch without the help of any frameworks.

The *vstplugin* project was commissioned in 2018 by the IEM in Graz. The initial idea was to "port" the recently developed IEM Ambisonic VST plugins (IEM 2021) to Pd and SuperCollider. The first alpha version was released in November 2018, the first stable 0.1 release in March 2019. By the time of writing, July 2021, the project has already reached version 0.5.1.

1. About VST plugins

1.1 History

The *Virtual Studio Technology* has been developed by Steinberg as a plugin interface for their digital audio workstation *Cubase*. The specification was released in 1996 together with *Cubase 3.02*. In 1999 Steinberg released the VST 2 SDK which introduced the concept of VST instruments (VSTi), such as software synthesizers or samplers, and added new features like MIDI events. Since then it has received several updates; the last version (2.4) was released in 2006.

In 2006 Steinberg introduced VST 3. The most important new features include multiple

input and output busses (e.g. for sidechaining or multi-timbral software instruments), sample accurate parameter automation, parameter groups and a resizable GUI editor. Because VST 3 is a completely different SDK and significantly more difficult to work with, plugin and host developers have been hesitant to adopt it. For them the new features did not seem to be worth the technical cost. In 2013 Steinberg officially declared the VST 2 SDK as deprecated, saying that they would not release any more updates.

TABLE 1 – VST 3 support in popular audio software in chronological order

Year	Program	Version	Company
2006	Cubase	4	Steinberg Media Technologies GmbH
2009	FL Studio	9.0.3	Image-Line Software
2013	Sonar	X3	Cakewalk Inc.
2014	Samplitude Pro	X2	Magix Software GmbH
2015	REAPER	5	Cockos Incorporated
2017	Bitwig Studio	2	Bitwig GmbH
2018	Max/MSP	8	Cycling '74
2019	Ableton Live	10	Ableton AG
2020	Ardour	6.5	Linux Audio Systems

This statement did not really impress host and plugin developers. In 2018, however, the situation changed dramatically when Steinberg announced that they would remove the VST 2 SDK and stop issuing new licenses. In practice, this meant that new plugin and host developers could not implement VST 2 support and were consequently forced to use the VST 3 SDK. Steinberg tried to purge copies of the SDK from the internet by sending DCMA requests. (The license explicitly forbids redistribution.) There exist several open source reimplementations whose legal status is somewhat unclear. Fortunately, since 2017 the VST 3 SDK is dual licensed: developers can choose between the "Proprietary Steinberg VST 3" license or the GPLv3.

1.2 Technical details

From a technical point of view, VST plugins are dynamic libraries (Windows DLLs, macOS bundles or Linux shared objects) that are explicitly loaded by the host with functions like

LoadLibrary or **dlopen**. The VST 2 SDK is a simple C plugin API. A plugin instance is a C struct with function pointers for processing and parameter automation. All additional operations are performed via a generic dispatcher function that takes an "opcode" (a number describing a certain operation) followed by up to four arguments. The plugin can also call into the host application via a callback function that is passed to the plugin's entry function when it is loaded by the host.

The VST 3 SDK is based on Steinberg's "VST Module Architecture" (Steinberg 2021) which is very similar to Microsoft's COM (Microsoft 2018), but significantly easier to work with. The basic idea is that related functionality is grouped in interfaces. On a low level, an interface is just an array of function pointers, but it can be conveniently represented in C++ as a pure abstract class, i.e. a class containing only virtual methods. (To ensure binary compatibility, such a class must not contain a virtual destructor or overloaded methods.) Additionally, each interface is associated with a unique identifier (UID). A UID is a 128-bit pseudo random number that can be created with the **uuidgen** command line tool, for example.

Every VST 3 interface inherits from an interface called **FUnknown** which contains the methods **addRef**, **release** and **queryInterface**. **addRef** increases the object's reference count and **release** decreases the reference count, destroying the object if the count has reached zero. **queryInterface** can be used to safely convert from one interface to another. In COM-like object models, extending functionality is generally not achieved by inheritance, instead a new interface is added. A VST 3 component generally inherits from all the interfaces it wants to support. If the client wants to obtain a different interface for a given object, it can pass the corresponding UID to the **queryInterface** method; if the interface is supported, the method returns a new pointer, otherwise it returns an error code.

As a consequence, it is possible for plugin and host developers to implement their own private or public extensions simply by adding new interfaces. Strictly speaking, such extensions were already possible in the VST 2 SDK with the **audioMasterVendorSpecific** and **effVendorSpecific** opcodes; one notable example are the REAPER VST 2.x extensions (Cockos 2021). VST 3, however, provides a much safer and convenient mechanism. A popular third party extension is *ARA* (Celemony 2021) by Celemony Software and PreSonus. It has been specifically developed for the *Melodyne* plugin, but has been used by a few other plugins since. ARA itself uses a pure C API, but it offers VST 3 interfaces that the plugin can implement to make the ARA functionality

available to the host.

Generally, a VST plugin is called from several threads. On the audio thread, the host calls the plugin's processing method. It might also send parameter changes (e.g. from parameter automation curves) and MIDI events. At the same time the user might automate parameters in the graphical editor, which runs on the UI thread. With the VST 2 SDK, it is the responsibility of the plugin to ensure that parameters can be automated in a thread-safe manner. Usually, this is achieved with atomic operations. The VST 3 SDK, on the other hand, enforces a clear separation between processor and editor. Developers are encouraged to implement them as separate components which do not share any state to eliminate the possibility of race conditions in the first place. Another advantage of this clear separation of concerns is that the components can run in different processes or even on different machines. It is the responsibility of the host to facilitate the communication between components. This is actually a good thing because it means that plugin developers do not have to worry about thread-safety. From an economical point of view, it makes sense to solve this problem once per host and not for every plugin. After all, there are orders of magnitudes more VST plugins than VST hosts.

2. Features

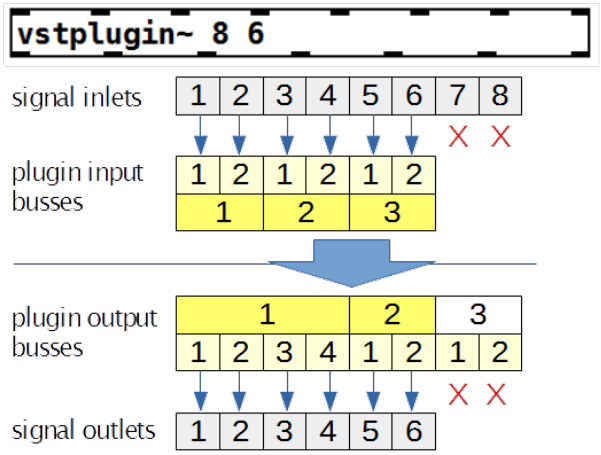
I will now present an overview over the various features that **[vstplugin~]** offers to the user. This section is not meant as an exhaustive reference. The full documentation can be found in **vstplugin~-help.pd**.

2.1 Input and output busses

By default, **[vstplugin~]** will create 2 input and 2 output channels. The user can provide the desired number of input and output channels as creation arguments. For example, **[vstplugin~ 4 8]** would create 4 input channels and 8 output channels. To be more specific, it creates a single input bus of 4 channels and a single output bus with 8 channels. VST 2 plugins only ever have a single input and output bus, but VST 3 plugins support several input or output busses. When the user opens a plugin, it receives the available input and output busses, so it can adjust its

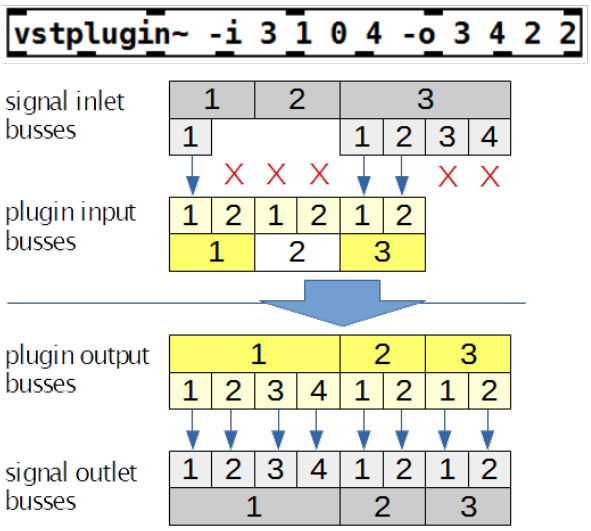
channel arrangements accordingly. For example, the IEM Ambisonic plugins can automatically infer the Ambisonic order from the number of available channels. If the plugin supports multiple input or output busses, channels are automatically distributed across consecutive busses.

FIGURE 1 – Automatic input/output channel distribution[1]



It is also possible to create several **[vstplugin~]** signal busses which are then mapped to the corresponding plugin busses. This can be achieved with the **-i** and **-o** creation argument flags. Each of them takes the number of input resp. output busses followed by the number of channels for each bus. If a bus has a channel count of zero, it is ignored.

FIGURE 2 – Manual input/output channel distribution



[1] All figures were prepared by the author.

2.2 Plugin search

The user can search for installed plugins with the **[search <flags> <paths>(** method, where **<paths>** is a list of the folders that it should search for plugins. If the list is empty, it automatically looks in the default VST plugin search paths.

TABLE 2 – Default VST plugin search paths

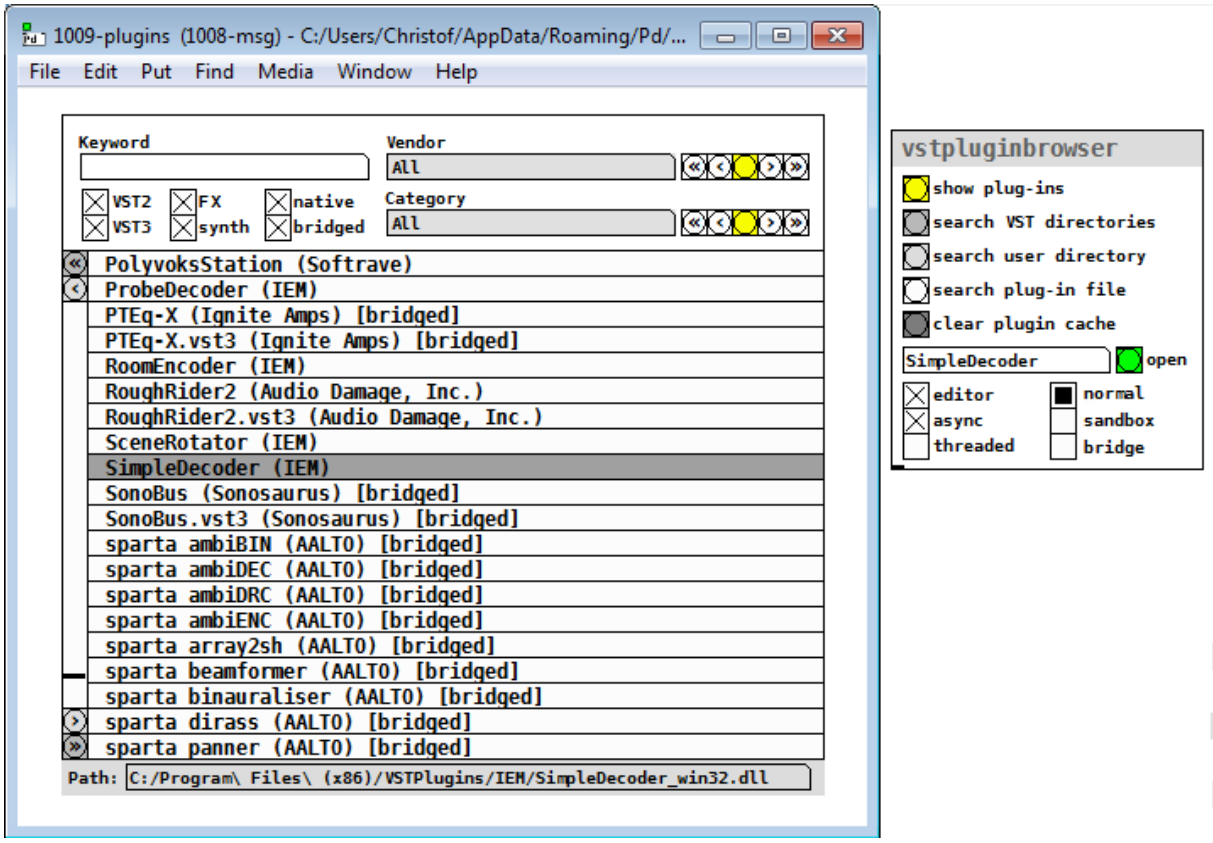
VST 2.x		**VST 3.x**	
Windows	%ProgramFiles%\VSTPlugins %ProgramFiles%\Steinberg\VSTPlugins %ProgramFiles%\Common Files\VST 2 %ProgramFiles%\Common Files\Steinberg\VST 2	**Windows**	%ProgramFiles%\Common Files\VST 3
macOS	~/Library/Audio/Plug-Ins/VST /Library/Audio/Plug-Ins/VST	**macOS**	/Library/Audio/Plug-Ins/VST 3 ~/Library/Audio/Plug-Ins/VST
Linux	~/.vst /usr/local/lib/vst /usr/lib/vst	**Linux**	~/.VST 3 /usr/local/lib/VST 3 /usr/lib/VST 3

Only the VST 3 folders are actually standardized, the VST 2 folders are just conventions.

The search results are stored in a dictionary where each plugin is associated with its name, so it can be easily referenced by the user. The dictionary is also written to disk as a cache file. The next time the user starts Pd and creates a **[vstplugin~]** instance, it would try to read the cache file and reconstruct the plugin dictionary. Depending on the plugins, the search can take quite a long time, but only the very first time. Each plugin is probed in a dedicated subprocess so that a plugin crash does not abort the search or terminate Pd.

For convenience I also provide an abstraction called **[vstpluginbrowser]** which shows all available plugins in a list. You can select a plugin and open it with several options (see 2.3). It is also possible to search the list and apply filters like SDK version (VST 2 or VST 3), plugin type (effect or instrument), plugin category, etc.

FIGURE 3 – The**[vstpluginbrowser]** abstraction



2.3 Open/close plugins

The **[open <flags> <name/path> <async>(** method opens a plugin either by its name or by its file path. The latter only works if the file contains a single plugin. VST 2 shell plugins (like *Waves*) and VST 3 plugin factories can contain multiple plugins in a single module. If **<async>** is 1, the plugin is opened asynchronously on a background thread; otherwise it is loaded synchronously. Either way, an **[open <name> <success>(** message is sent when the operation has finished. Additionally, you can provide the following flags:

-e: use the plugin GUI editor (see 2.4)

-t: process the plugin in a separate thread (see 2.10)

-p: run the plugin in a dedicated subprocess (see 2.9)

-b: run the plugin in a shared subprocess (see 2.9)

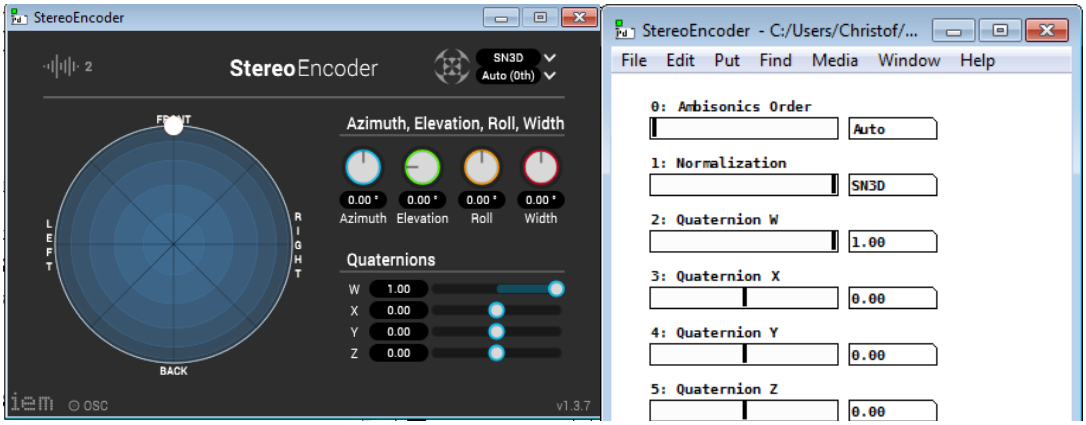
2.4 Plugin editor

[vstplugin~] provides two kinds of editors:

a) the plugin GUI editor

b) a generic Pd editor

FIGURE 4 – plugin GUI editor vs. generic Pd editor



You can open the editor window simply by clicking into the plugin object. Alternatively, you can use the **[vis 1(** method. The window position can be set programmatically with the **[pos <x> <y>(** method. The window size can be changed with the **[size <w> <h>(** method, but this is only works for plugins which actually have a resizable editor.

There is a severe limitation on macOS: the Cocoa framework mandates that the UI must run on the main thread – which happens to be the audio thread in Pd. You can use the plugin GUI editor, but it can easily cause audio dropouts. Therefore, you are advised to keep the editor window closed whenever you do not need it. Note that this does not affect plugins that run in a subprocess, so a possible workaround is to open the plugin with the **-p** or **-b** flag (see 2.3).

2.5 Parameter automation

A VST parameter is a numeric value that describes a certain aspect of a plugin, like output volume, filter frequency, compressing ratio, etc. It is typically associated with a GUI control so that it can be directly manipulated by the user. Parameters can also be changed programmatically, like in

parameter automation curves as used in practically all DAWs.

Parameters can be continuous, step wise or binary, but internally they are always implemented as floating point numbers. They can be represented in three different ways:

1. normalized, always 0.0 – 1.0
2. plain (e.g. 20 – 20000)
3. string (e.g. “on”, “off”)

In **[vstplugin~]** I only allow to set and get parameters as normalized floats and strings. The latter is not supported by all plugins resp. it might not work for all parameters. I have not implemented “plain” parameter values because they are not fully implemented in the VST 2 SDK.

Each parameter has an index, a name (e.g. “Volume”) and an optional label (e.g. “dB”). The **[param_count(** method yields the number of available parameters; the **[param_info(** method outputs the name and label of the parameter at the given index. In the VST 2 SDK, the index of a parameter simply corresponds to its position in the parameter list. In the VST 3 SDK, each parameter is actually associated with an arbitrary ID, but for the sake of consistency we use the parameter index instead. With the **[param_set <which> <value>(** method, users can set a parameter to a given value. **<which>** can be an index (float) or name (symbol); **<value>** can be a normalized float or a string (symbol). Conversely, **[param_get <which>(** outputs the normalized float value and string representation of the given parameter.

With the VST 2 SDK, parameter changes contain no timing information, so they are effectively aligned to process block boundaries. The VST 3 SDK, on the other hand, introduced sample accurate parameter automation. Each parameter change has a sample offset relative to the current block; it is even possible to have multiple parameter “points” per block. However, because Pd patches usually run at a very low block size (64 samples) we currently only support a single parameter point per block – albeit with the correct sample offset. Actually, there is a hidden compile time option for the *vstplugin* project to enable multi-point parameter automation, but it is disabled by default to save CPU and memory. Note that only few VST plugins actually seem to implement sample-accurate parameter automation at all! The popular JUCE framework, for example, does not support it (at the time of writing).

2.6 Preset management

Every DAW allows to save a project and open it at a later time. This means that it also has to save and restore the state of every plugin. One might think that it is sufficient to simply iterate over all parameters and save their values. In a way, this is how many simple plugins work. However, plugins can also have additional state that is not exposed via parameters. Sampler instruments are a common example, as they also need to store and recall the set of audio samples. As a consequence, VST plugins provide methods to set and get the complete plugin state as a binary blob.

Program files

Hosts can take the data and store it in one of the standard preset file formats (**.fxp** for VST 2 plugins and **.vstpreset** for VST 3 plugins). This can be achieved with the methods **[program_write <filename> <async>(** and **[program_read <filename> <async>(**. The **<async>** argument determines whether the operation should happen synchronously (0) or asynchronously (1). The advantage of the latter is that everything happens on a background thread, so it does not block the audio thread. The processing of the plugin itself, however, might have to be temporarily suspended.

Program data

Alternatively, hosts might wish to integrate the plugin state into their own preset management system. For this purpose, the **[program_data_get(** method outputs the plugin state as a Pd list of raw bytes; the state can be restored by passing the list to the **[program_data_set(** method. Pd users could use it in conjunction with the [text] object to implement their own preset lists.

Presets

One disadvantage of the methods above is that the user has to think where exactly they want to store the plugin state on the file system. For this reason, **[vstplugin~]** provides the methods **[preset_save(and [preset_load(**. They basically work the same as the **[preset_write(and [preset_read(** methods, but instead of passing a file path, the user provides a preset name. Behind the scenes, an appropriate file name is generated and the preset is stored in a well defined location. Since the location of VST 3 presets is actually standardized, presets can be automatically exchanged between different hosts.

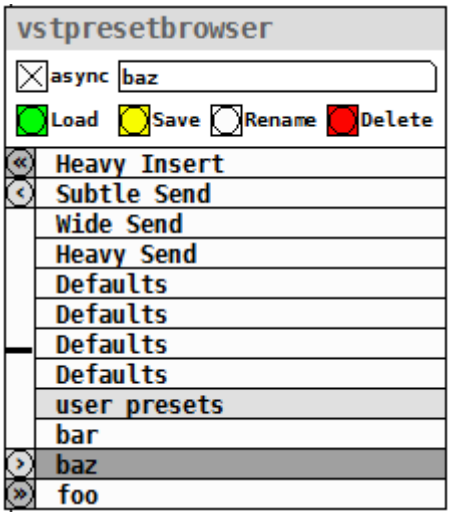
It is worth noting that quite a few plugins implement their own preset management system with custom file formats and folders. These systems are generally only accessible via the plugin GUI editor and not really integrated into the host environment. However, it is always possible to save and restore the complete plugin state with the methods listed above, which means that you can effectively "convert" presets from one format to the other.

Built-in programs

Additionally, VST plugins might have a list of built-in programs. These can be selected with the **[program_set <index>(** method. Most of the time, these programs are immutable, but some VST 2 plugins allow to manipulate each of them and save the whole list as a so called "bank file" (**.fxb**). This feature is not commonly used because hosts tend to create their own preset lists. However, it is supported via the **[bank_write(** and **[bank_read(** resp. **[bank_data_get(** and **[bank_data_set(** methods.

[vstplugin~] ships with a useful abstraction called **[vstpresetbrowser]**. It displays a list of available built-in programs and user generated presets. Users can connect it to a **[vstplugin~]** instance to save the current state as a new preset or load an existing preset. It is also possible to rename and delete presets. If the preset list is modified in any way, all related **[vstpresetbrowser]** instances are automatically updated.

FIGURE 5 – The **[vstpresetbrowser]** abstraction



2.7 MIDI

MIDI messages are most commonly used to play VST instruments. However, there are also effect plugins that react to MIDI events. Finally, there is a whole category of MIDI plugins which process, filter or produce MIDI message and do not produce any sound at all. You can send raw MIDI messages with the **[midi_raw <bytes>(** method. In addition, we also provide the following convenience methods:

TABLE 3 – MIDI methods

[midi_note <chn> <pitch> <velocity(	Note-on message. A few plugins support microtonal playback in which case **<pitch>** can be a fractional value!
[midi_noteoff <chn> <pitch> <velocity>(	Note-off message.
[midi_cc <chn> <number> <value>(	CC message.
[midi_bend <chn> <value>(	Pitch bend message. The range for **<value>** is -1 to 1.
[midi_program <chn> <program>(	Program change message.
[midi_polytouch <chn> <pitch> <pressure>(	Polyphonic aftertouch message.
[midi_touch <chn> <pressure>(	Channel aftertouch message.

The range for **<chn>** (MIDI channel) is 1–16.

It is also possible to send SysEx messages with **[midi_sysex <bytes>(**. This feature is sometimes used in plugins that emulate old hardware devices, such as the Yamaha DX7 synthesizer, so people can use existing patches. A plugin can also produce MIDI or SysEx events. These are translated to the Pd messages **[midi <bytes>(** resp. **[sysex <bytes>(** and sent through the rightmost outlet.

In the VST 2 and VST 3 SDK, MIDI messages contain a sample offset relative to the current process block. This is necessary to maintain accurate timing with large block sizes. For example, a block size of 1024 at 44.1 kHz has a duration of 23 milliseconds. If MIDI messages would be aligned to block boundaries, the resulting jitter would be unacceptable. Although Pd patches typically run at very low block sizes (64 samples), we always schedule MIDI messages with the appropriate sample offset.

2.8 Timing and transport

Some plugins depend on additional timing information to work correctly. For example, sequencers and arpeggiators need to know the musical tempo and current bar position to run in sync with the host. Tempo information can also be used to convert musical units to actual time values. For example, a delay plugin might allow to set the delay time in terms of note values; internally, the actual delay time can be calculated based on the current musical tempo. For this purpose, the VST SDK defines a structure that contains information such as the elapsed time, the current position in the project in musical beats, the last bar start, the current tempo, the current time signature, the MIDI clock offset, the SMPTE frame offset, etc. This structure is maintained by the host and updated regularly. It is provided to each plugin during processing, so it can retrieve the information it needs.

A DAW typically operates on a single global timeline, which means that at any given time all plugins observe the same time signature, musical tempo and bar position. However, this is a major limitation that many people who use computer music environments like Pure Data or SuperCollider have tried to escape in the first place. For this reason, each instance of **[vstplugin~]** maintains its own time context structure which can be manipulated independently. Of course, you can still run several instances in sync simply by passing messages to all of them at the same time.

Most of the time context members are updated automatically and do not need to be exposed to the user. Others could be exposed but do not make much sense in the context of Pd. The following methods are currently available:

TABLE 4– Timing and transport methods

[tempo <bpm>(	Set the tempo in beats per minute. The default is 120 BPM.
[time_signature <num> <denom>(	Set the time signature. For example, **[time_signature 3 4(** means 3/4. The default is 4/4.
[play <bool>(	Start/stop the transport. Note that transport is off by default!
[transport_set <beat>(	Set the current transport position in quarter note beats (can be fractional).
[transport_get(	Get the current transport position.

2.9 Bridging and sandboxing

Generally, plugins have to be compiled for the same CPU architecture as the host application. However, there are thousands of existing VST plugins that only exist as 32-bit Intel binaries and will never be updated to 64-bit because the company has discontinued the plugin or does not exist anymore. Most of the time, the source code is not available, so you cannot simply recompile it. When DAW manufacturers updated their products to 64-bit they tried to make sure that their users can still run their existing 32-bit plugins. This process is called "bit bridging" and it can be achieved with various means. All the approaches have one thing in common: the plugin runs in a separate 32-bit process that somehow communicates with the DAW. Of course, it is also possible to achieve the opposite (using a 64-bit plugin from a 32-bit host application), but this is not as common.

Bit bridging is mainly used on Windows because 32-bit and 64-bit software largely coexist up this day. On macOS the situation has changed fundamentally since macOS 10.15 Catalina has completely removed 32-bit support. However, the advent of the M1 ARM chips could mean that bit bridging might become useful again: Although it is possible to run existing x86-64 apps on ARM via *Rosetta* (Apple 2021), an app compiled for ARM still cannot load x86-64 plugins. This can be solved by running the plugin in a x86-64 subprocess that communicates with the ARM host

application.

On Linux, traditional bit-bridging is not very common because plugins are either open source or they are recent enough to provide 64-bit versions. However, the bridging technique can also be used to load Windows VST plugins (both 32-bit and 64-bit) in a Linux host application by running the subprocess in *Wine* (WineHQ 2021).

[vstplugin~] v0.4 introduced built-in support for bit-bridging. The source code contains a separate application that can be compiled for various CPU architectures. Since v0.5 the bridge application can also be built for Wine. **[vstplugin~]** always prefers native plugins over bridged plugin with the same name. Plugins that have been "converted" with an external plugin bridge, like jBridge, LinVst or Yabridge, will appear as native plugins.

Naturally, it is also possible to run native plugins in a separate process. The main advantages are increased safety and stability. If a plugin accidentally crashes, the host application is not negatively affected ("sandboxing"); most importantly, the user does not lose their work because of a program crash. For 32-bit applications it is also a way to increase the available memory space. A 32-bit app can only use up to 4 GB of RAM, some of which is reserved by the OS, and sampler instruments can easily reach this limit. The disadvantages are increased memory usage, some CPU overhead and possible rescheduling because of the necessary synchronization between the two processes.

For the purpose of bit bridging, it is sufficient that all plugins of the same CPU architecture share the same subprocess. For sandboxing there is one severe disadvantage: if one plugin crashes, it will take down all other plugins in the same process. Therefore it might make sense to run a plugin in a dedicated process – at the cost of additional overhead. Some DAWs let you decide for each plugin whether you want to run it a) directly in the host, b) in a shared process or c) in a dedicated process. With **[vstplugin~]** this can be done with the -p and -b flags for the **[open(** method (see 3.2).

2.10 Multithreading

CPU clock speed has been more or less stagnated for the last 15 years for various reasons; instead, modern CPUs contain multiple cores that can execute programs in parallel. Pure Data,

however, has been designed in the mid 90s where CPU clock speed was still increasing rapidly and multicore CPUs did not exist yet. As a consequence, its architecture is fundamentally single threaded. While the GUI runs in a separate process and certain tasks, like disk streaming, are now done on separate threads, audio processing is still restricted to a single thread.

[vstplugin~] allows to leverage the possibilites of multicore CPUs by processing plugins on different threads, keeping the main audio thread free for other tasks. Internally, it manages a thread pool of DSP worker threads. Multithreading can be enabled per plugin by opening it with the **-t** flag (see 2.3). Because of technical limitations, multithreading introduces a delay of 1 block per plugin!

2.11 Offline processing

This section is not about the VST 2 offline processing interface – an obscure part of the VST 2 SDK that has been removed entirely from the VST 3 SDK; instead, it is about the difference between real-time processing, like recording and playback, and non-real-time (offline) processing, like rendering to disk. In a real-time context, plugins have to make sure to process audio as fast and evenly as possible to avoid audio dropouts. In a non-real-time context, however, they do not have this limitation, so they can decide to render at a higher quality and employ algorithms that are more costly or cause CPU spikes. Also, I have witnessed at least one plugin that would not work correctly at all if not being informed about offline processing.

Pd is mostly used in real-time interactive mode, but it is also possible to run it in batch mode by starting Pd with the **-batch** option. Additionally, Pd 0.52 introduced the "fast-forward" message:

> You can send "pd fast-forward" to make Pd run its scheduler at full speed, without waiting for A/D/A or the real-time clock. Useful for running batch-style jobs interactively. You can advance a variable amount of time by specifying a large value and then later sending "pd fast-forward 0" - but note that outside input is ignored while in fast forward. (Puckette 2020)

With the **[offline <bool>(** method you can inform the plugin about the current processing mode.

Conclusion

[vstplugin~] is a powerful cross-platform tool for hosting VST plugins of all sorts. It implements almost all of VST 2.x and the most important parts of VST 3. While VST 2.x has not been updated for over 14 years (and is officially deprecated), VST 3 is continuously expanding its feature set. Generally, I only try to implement features that are useful in the context of a computer music environment like Pd. Large parts of the VST 3 SDK, such as parameter groups, context menus, speaker names, preset categories, MIDI learn, etc., are specifically aimed at DAWs. Many of these features are purely informative and do not provide actual functionality.

Of course, there is still room for improvement. For example, the VST 3 SDK currently has poor multichannel support. (It is not possible to have more than 24 channels per bus.) If Steinberg decides to fix this, by releasing a new interface, I will update **[vstplugin~]** accordingly. Also, I can imagine adding ARA support once more plugins start using it. At the moment, however, I consider the project more or less feature complete.

ACKNOWLEDGMENT

I would like to thank the IEM for initiating and supporting this project and all the people who have tested the external, reported bugs and gave feedback.

REFERENCES

APPLE. *About the Rosetta Translation Environment.* 2021. Accessed July 15, 2021. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment

CELEMONY. *ARA SDK code repository*. 2021. Accessed July 15, 2021. https://github.com/Celemony/ARA_SDK

COCKOS. *Cockos Extensions to VST SDK*. 2021. Accessed July 15, 2021. https://www.reaper.fm/sdk/vst/vst_ext.php

GRILL, Thomas. v*st~ code repository*. 2009. Accessed July 15, 2021. https://github.com/grrrr/vst

IEM. *Ambisonic plugins*. 2021. Accessed July 15, 2021. https://plugins.iem.at/

MICROSOFT. *Component Object Model*. 2018. Accessed July 15, 2021. https://docs.microsoft.com/en-us/windows/win32/com/the-component-object-model

PUCKETTE, Miller. *Pd documentation. 5.1 release notes.* 2020. Accessed July 15, 2021. http://msp.ucsd.edu/Pd_documentation/x5.htm#s1

RESSI, Christof. v*stplugin code repository*. 2021. Accessed July 15, 2021. https://git.iem.at/pd/vstplugin/

STEINBERG. *VST Module Architecture*. 2021. Accessed July 15, 2021. https://developer.steinberg.help/display/VST/VST+Module+Architecture

WINEHQ. *WineHQ homepage*. 2021. Accessed July 15, 2021. https://www.winehq.org/

ABOUT THE AUTHOR

Christof Ressi is an Austrian composer, arranger, software developer and multimedia artist. He studied composition, jazz arrangement and computer music at the University of Arts in Graz (Austria). His music has been performed all over the world, he has won several prizes and his computer music work has been featured at several international festivals and conferences. 2017–2019 he has been part of the artistic research project *GAPPP* at the Institute for Electronic Music (IEM) in Graz (Austria). He is currently pursuing his artistic doctoral studies at the Anton Bruckner Privatuniverstität in Linz (Austria). ORCID: https://orcid.org/0000-0001-8818-0306. E-mail: info@christofressi.com